DNOS

TI Pascal Programmer's Guide

Part No. 2270517-9701 *A
January 1984

TEXAS INSTRUMENTS

Printed in U.S.A.

MANUAL REVISION HISTORY

DNOS TI Pascal Programmer's Guide (2270517-9701)

Original Issue . . . . . 1 August 1981
Revision . . . . . January 1984

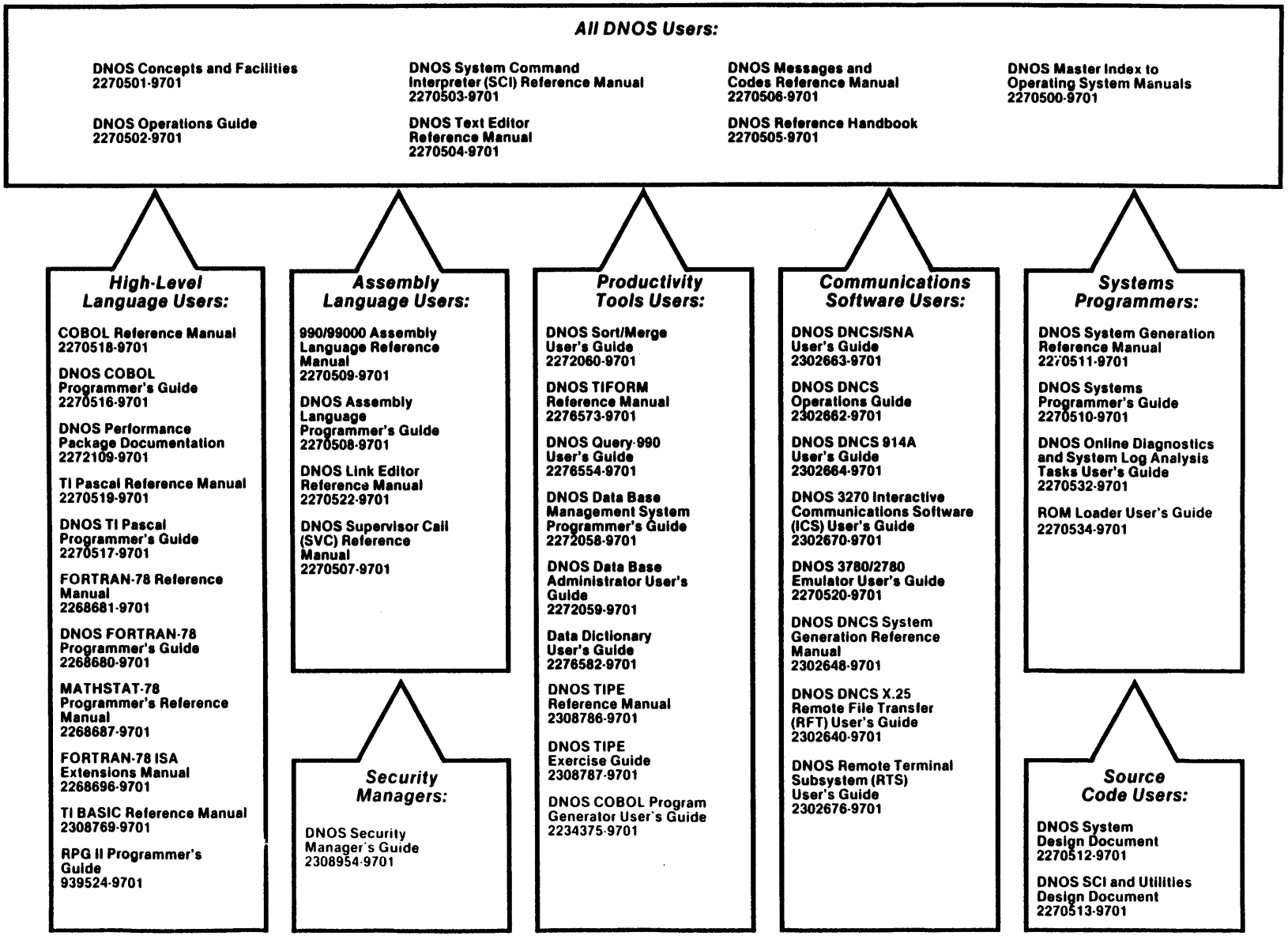
The total number of pages in this publication is 356.

The computers, as well as the programs that TI has created to use with them, are tools that can help people better manage the information used in their business; but tools—including TI computers—cannot replace sound judgment nor make the manager's business decisions.

Consequently, TI cannot warrant that its systems are suitable for any specific customer application. The manager must rely on judgment of what is best for his or her business.

DNOS Software Manuals

This diagram shows the manuals supporting DNOS, arranged according to user type. Refer to the block identified by your user group and all blocks above that set to determine which manuals are most beneficial to your needs.



DNOS Software Manuals Summary

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

DNOS Operations Guide
Explains fundamental operations for a DNOS system. Includes detailed instructions on how to use each device supported by DNOS.

System Command Interpreter (SCI) Reference Manual
Describes how to use SCI in both interactive and batch jobs. Describes command procedures and gives a detailed presentation of all SCI commands in alphabetical order for easy reference.

Text Editor Reference Manual
Explains how to use the Text Editor on DNOS and describes each of the editing commands.

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

DNOS Reference Handbook
Provides a summary of commonly used information for quick reference.

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
Contain information about the languages supported by DNOS. Each programmer's guide covers operating system information relevant to the use of that language on DNOS. Each reference manual covers details of the language itself, including language syntax and programming considerations.

Performance Package Documentation
Describes the enhanced capabilities that the DNOS Performance Package provides on the Model 990/12 Computer and Business System 800.

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.

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

DNOS System Generation Reference Manual
Explains how to generate a DNOS system for your particular configuration and environment.

User's Guides for Productivity Tools
Describe the features, functions, and use of each productivity tool supported by DNOS.

User's Guides for Communications Software
Describe the features, functions, and use of the communications software available for execution under DNOS.

Systems Programmer's Guide
Discusses the DNOS subsystems and how to modify the system for specific application environments.

Online Diagnostics and System Log Analysis Tasks User's Guide
Explains how to execute the online diagnostic tasks and the system log analysis task and how to interpret the results.

ROM Loader User's Guide
Explains how to load the operating system using the ROM loader and describes the error conditions.

DNOS Design Documents
Contain design information about the DNOS system, SCI, and the utilities.

DNOS Security Manager's Guide
Describes the file access security features available with DNOS.

Preface

This manual contains information about TI Pascal (TIP), which executes on the Texas Instruments Business Systems Computers. This information supports the experienced programmer in developing TIP application programs intended for execution under the DNOS operating system. For additional descriptions of TIP, refer to the *TI Pascal Reference Manual*.

This manual contains the following sections and appendixes:

Section

1 Introduction — Gives a brief overview of DNOS and introduces the steps involved in developing a TIP program. Explains notations that describe commands appearing throughout this manual.

2 DNOS Concepts and Environment — Describes DNOS features related to program development.

3 Building a TI Pascal Program — Discusses how to build a TIP program, beginning with directory and file development, and discusses how to use the Text Editor.

4 Nester Utility — Contains information about the Nester utility, which formats source code in a consistent block structure.

5 TIP Compiler — Discusses in detail the TIP compiler execution and options.

6 Separate Compilation — Examines separate compilation of TIP program modules. This section describes the configuration processor, the Split Program utility, and the Split Object utility, all of which assist in separate compilation. The discussions include options and commands for each process.

7 Link Editing and Execution — Describes options for link editing TIP tasks, executing TIP tasks, and TIP run-time options.

8 Internal Structures — Discusses stack and heap requirements and internal data structures.

9 Debugging — Describes the SCI debug capabilities and commands and the TIP debug commands. Also describes the types of run-time errors and abnormal termination dumps provided.

10 Run-Time Library Routines — Describes the routines that a user program can call for direct communications register unit (CRU) I/O and operating system interface.

11 Assembly Language Routines — Describes the interface required for routines written in assembly language, including task termination routines.

12 Interfacing to Productivity Tools — Examines the requirements for interfacing to TIFORM, DBMS-990, Query-990, Sort/Merge.

13 Reverse Assembler (RASS) Utility — Describes the reverse assembler and its output.

Appendix

A Keycap Cross-Reference — Charts the corresponding key on available terminals for each generic keyname, and illustrates each terminal's keyboard.

B TIP Compiler Error Messages and Codes — Lists TIP compiler error messages and codes.

C Run-Time Error Messages and Codes — Describes the types of TIP run-time errors, and lists TIP run-time error messages and codes.

D Estimating Run-Time Sizes — Discusses the run-time size needed for various program features.

In addition to the DNOS manuals shown on the frontispiece, the following documents contain additional information related to TIP:

Title	Part Number
TI Pascal Reference Manual	2270519-9701
TI Pascal Configuration Processor Tutorial	2250098-9701
DNOS TI Pascal Object Installation	2276550-9701
TIFORM Reference Manual	2234391-9701
Guide to the TI Pascal Run-Time Support System	2250035-9701*

* Design Document. Available only with the purchase of *TI Pascal Run-Time Source*.

Contents

3 — Building a TI Pascal Program

6 — Separate Compilation

7 — Link Editing and Execution

8 — Internal Structures

9 — Debugging

10 — Run-Time Library Routines

11 — Assembly Language Routines

12 — Interfacing to Productivity Tools

13 — Reverse Assembler (RASS) Utility Details

Appendixes

Index

Illustrations

Tables

1

Introduction

1.1 GENERAL

This section provides an overview of the Texas Instruments DNOS operating system and the process of program development using the TI Pascal (TIP) programming language. During the preparation of this manual, some assumptions have been made for the sake of a clear presentation. You, the user, are assumed to have a running DNOS system with the System Command Interpreter (SCI), a video display terminal (VDT) in VDT mode, and a valid user ID and passcode. The terminal mode is typically set by a system manager at the installation. In VDT mode, all command prompts appear on the VDT screen at once; in TTY (teletype) mode, only one command prompt appears at a time. Usually, the system manager assigns your user ID and passcode, which are used to log onto the system.

1.2 DNOS OVERVIEW

DNOS is a general-purpose, multitasking operating system that operates with the TI 990/10, 990/10A, and 990/12 minicomputers. It is a versatile, disk-based operating system that supports a wide range of commercial and industrial applications. As a multiterminal system, DNOS can support several users, each of whom appear to have exclusive control of the system. DNOS includes the following features and capabilities:

- High-level language support (including TIP and COBOL)
- Job-level and task-level operations to facilitate the use of system resources
- Foreground, background, and batch processing
- A text editor
- An optional accounting function to collect and store resource utilization data
- A macro assembler
- A file management package that supports key indexed, sequential, and relative record files
- Output data spooling
- Interprocess (task-to-task) communication (IPC)
- Productivity tools for data base management, video display forms design, Query, and Sort/Merge operations

To operate the system, you can use a terminal to enter commands that are interpreted by SCI, or you can interact directly with application programs. As a result of the wide range of support functions, together with the available hardware, DNOS is well suited for both large-scale and small-scale applications. The *DNOS Concepts and Facilities Manual* (see the Preface) contains a more detailed overview of DNOS.

1.3 A TIP PROGRAM DEVELOPMENT OVERVIEW

TIP programs are usually entered into the computer via the interactive Text Editor. The program is called source code at this point, and the file created with the Text Editor is the program source file. (Programs prepared on external media, such as punched cards, may be read into the computer if the necessary peripheral devices are available.) At your option, a source file may be run through the Nester utility to improve the program's readability and to do a rapid check of the syntax. The Nester output is the source code indented on a standard format, emphasizing the inherent block structure of TIP. Either the original (text edited) source file or the nested source file serves as input to the TIP compiler.

The TIP compiler consists of six separate tasks, or stages. Together, these tasks detect syntax errors and semantic errors in the source code, translate the source code to object code, perform optimization functions, write the object code to an object file, and write a source code listing with errors to a listing file. When a program has been compiled without errors, the object file (output from the compiler) becomes the input file for the Link Editor.

The Link Editor ties program segments together with a TIP run-time library of standard procedures, producing linked object code. The linked object code may be output as an executable program and automatically installed on a program file.

The next step is to execute the program and debug it. Although good design and initial coding can minimize the time spent debugging, this is often the most time consuming phase of program development. If errors occur in the program, use the tools in the system Debugger to identify them. Correct the program at the source code level, and then compile and link it again to produce a working program.

1.4 COMMAND FORMAT AND NOTATION

This manual uses a standard format and notation to describe system commands. The notation reflects the conventions used in the *DNOS System Command Interpreter (SCI) Reference Manual*. Each command description shows the command keyword (usually an abbreviation of the command function) and the full command name along with any associated command messages or prompts. The following paragraphs explain these components.

1.4.1 Command Name

The command keyword represents the full command name. For example, Show Date and Time is a command name, and the command keyword is SDT. You enter SDT and press Return to execute the Show Date and Time command. The system responds by displaying the following:

```
SHOW DATE AND TIME
13:48:30 WEDNESDAY, JANUARY 25, 1984.:
```

Since this command includes no command prompts, the command executes without further user interaction.

1.4.2 Command Prompts

Upon entry of a command, the system displays the full name of the command and any associated command prompts. The prompts provide you with information and request responses from you to complete the execution of the command. Respond to the prompts as required. For example, when you enter the Show File (SF) command, the cursor appears after the FILE PATHNAME prompt. The system waits for you to enter a file pathname, such as .MYFILE. (A pathname is a character string that indicates a path to a resource such as a file, channel, or device.) Following your response to the first prompt of a command, press the Return key. The cursor moves to the field following the next prompt (if one exists) and awaits your response. After responding to the last prompt, press the Return key to activate the command. To cancel the command at any time, press the Command key.

1.4.3 Type of Response Expected

For each command prompt, a response of a given type is expected. In the remainder of this manual, the expected response type is shown after each command prompt. For example, in the SF command, the expected response type for the first (and only) prompt is a pathname:

```
[] SF
SHOW FILE
      FILE PATHNAME:  pathname
```

1.4.3.1 Initial Values. To help you respond to the prompts, the system sometimes displays an initial value automatically after a prompt or provides a default value. An initial value is a value that the system automatically displays as a response to some command prompts. You can accept an initial value by pressing the Return key, erase the initial value by pressing the Erase Field or Skip key, or replace the initial value by entering a different value.

The initial values for some prompts are fixed. For these prompts, the same initial value always appears. In other cases, the system saves a value entered with a command and displays it as an initial value for a later entry of the same command or for the entry of a related command. Some variable initial values are also saved from one terminal session to another.

1.4.3.2 Default Values. A default value is a value that the system automatically supplies as the response to a prompt when you do not enter a value. Often, the system provides default values to speed up the entry of responses to command prompts. This is especially true for optional user responses. To use the default value for a command prompt (where a default value exists), press the Return key without entering any other data. Such an entry is called a null entry.

1.4.4 Notation Symbols

Notation symbols, as shown in Table 1-1, enclose certain prompt responses in the command descriptions to help explain how the response is to be entered.

Table 1-1. Command 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 are separated by a slash).
item. . . item	More than one item of this type may be entered in response to the prompt. Items should be separated by commas.
@	Synonyms or logical names are allowed as responses (synonyms and logical names are described in Section 2).
( )	Represents the initial value. If (*) is shown, the value may be supplied from a synonym set by a previously used command. If a list is supplied in a form other than interactively (batch mode or a command calling a command), the list must be enclosed in parentheses.

1.5 SYNTAX NOTATION

The syntax of a programming language describes the form that a legal program in that language can take. In TIP, the syntax can be expressed concisely by either syntax diagrams or by the more traditional Backus-Naur Form (BNF), sometimes called Backus-Normal form.

1.5.1 Backus-Naur Form (BNF)

In BNF, each element of the language is defined by means of an equation-like rule called a *production*. The entity being defined is written to the left of the symbol ::= and the definition is written to the right of that symbol. The definition can be expressed in terms of language elements defined by additional productions. The following symbols are used in writing definitions:

Symbol	Purpose
::=	Writes productions; means "is defined to be"
< >	Encloses nonterminal symbols (i.e., entities defined by a production)
[ ]	Encloses optional entities
|	Represents alternatives (e.g., A | B | C means A or B or C)

NOTE

Both brackets ([ ]) and braces ({ }) are used in TIP as terminal symbols. When used in BNF productions to specify terminal symbols, the brackets and braces are enclosed in quotation marks.

An identifier may be defined as follows:

```
<identifier> ::= <letter> { <id character> }
<id character> ::= <letter> | <digit> | _(underscore)
<letter> ::= A | B | C | D |. . .| Z | $
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

In this manual BNF productions specify the language syntax. Syntax diagrams supplement the BNF productions to illustrate the syntax of TIP declarations and statements.

1.5.2 Syntax Diagrams

A syntax diagram is a directed graph with a single input edge and a single output edge. The graph represents a syntax rule. Any possible path from the input edge to the output edge corresponds to an application of the syntax rule. Figure 1-1 shows the symbols that appear in syntax diagrams and provides an example of a syntax diagram for an identifier.

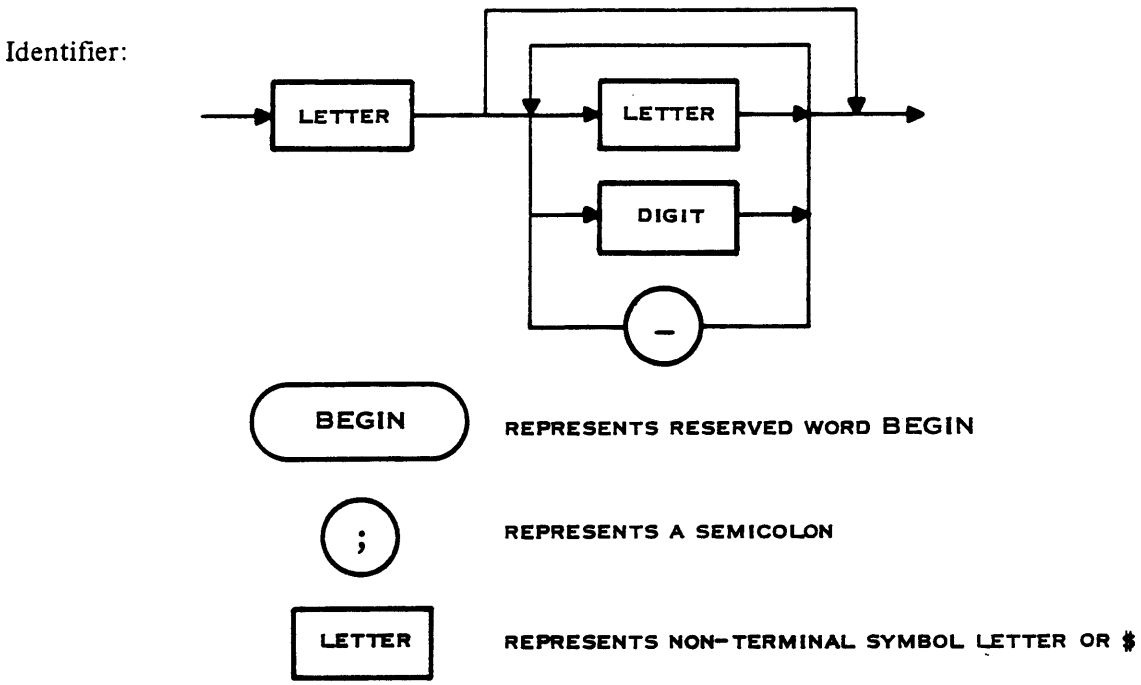


Figure 1-1. Syntax Diagram Symbols

1.6 CHARACTER SET

The TIP character set consists of the letters A through Z, the digits 0 through 9, and the following special characters:

+ - * / " . , ; : = ' < > ( ) [ ] { } # ∧ @ ? _ $

Lowercase letters can be used on VDT's and other devices that have both uppercase and lowercase letters. However, the TIP compiler translates these letters to uppercase. Consequently, the reserved word BEGIN can be entered as BEGIN, Begin, or begin. Also, identifiers MYPROG, Myprog, and myprog are not unique. The compiler processes any one of them as if it were MYPROG.

The characters are used to form symbols that have a fixed meaning in the language. Some of these special symbols are used for operators and delimiters:

```
+  -  *  /  :=  =  <>  <  <=  >=  >  ::  ()  (..)  (**)  ..  .  ,  ;  :  '  "  #  @  ?
```

NOTE

To delimit array indices and sets, (..) may be substituted for [ ]. To delimit comments, (**) may be substituted for { }. To identify pointers, @ may be substituted for ^.

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 Fortran Task (XFT), 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 Systems Programmer's Guide* 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 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. If you set the lowercase option with the .OPTION primitive, you can use either upper or lowercase characters. 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:
```

2.3.3 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.

2.3.3.1 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
```

2.3.3.2 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.
```

2.3.4 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.

2.3.4.1 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
```

2.3.4.2 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 *DNOS 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
              PAGE LENGTH: 59
```

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
              PAGE LENGTH: 59
```

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 PRINT WIDTH (CHARS) and PAGE LENGTH prompts are accepted by omitting them 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.

2.3.4.3 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 Command 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 execute a batch SCI job independent of a terminal. Consequently, you can continue to execute SCI commands in foreground or background mode. The *DNOS System Command Interpreter (SCI) Reference Manual* contains a description of the XBJ command. The XBJ command is very similar to the XB command if you start the batch job at your local site with your own user ID.

2.3.4.4 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 2-1).

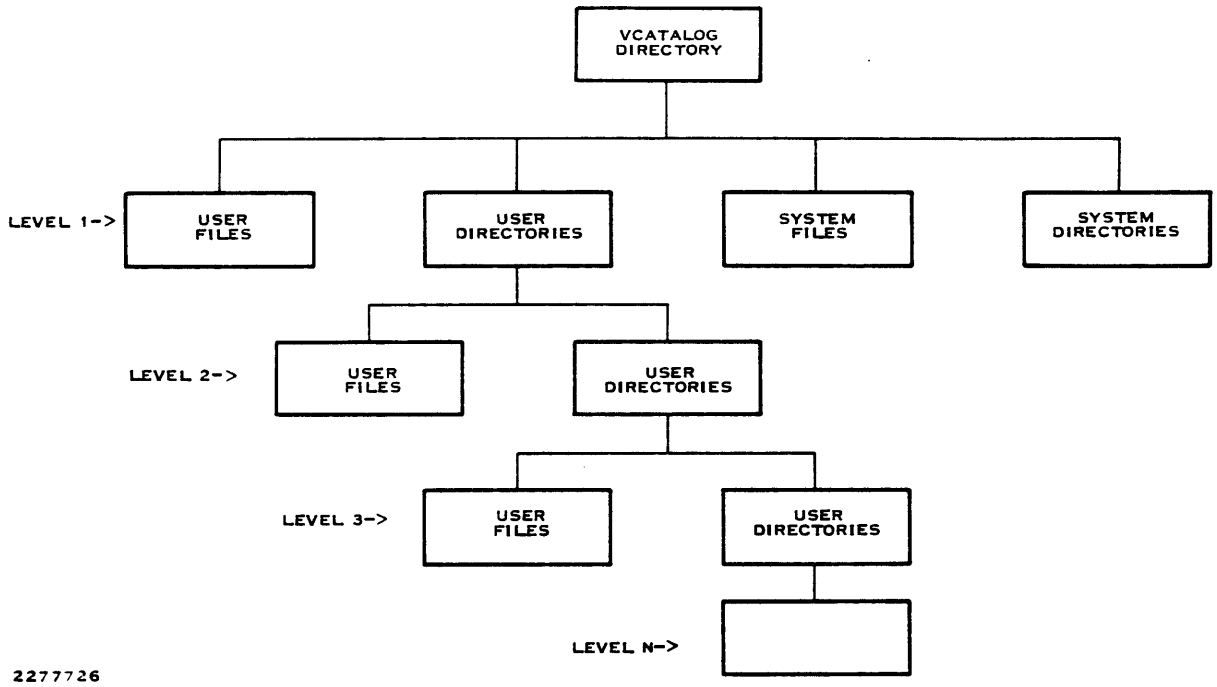


Figure 2-1. Directory and File Structure

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.

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.

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 the periods used to separate directories, subdirectories, and files. 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
DS01	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

When you use DNOS in a network of DNOS systems, you can access files and devices at any site in the network by using a site name and a colon as the first part of the access name. For instance, to use LP02 at a site named Dallas, you can specify DALLAS:LP02. To access the file DS02.S$NEWS at a site named Dallas, you can specify DALLAS:DS02.S$NEWS. The 48 character limit for file names includes the site name if you specify one.

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

DNOS supports use of synonyms for I/O resources. *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 is 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. This 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 set of files, in sequence, without physically concatenating the files. (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 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 can not 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 record 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 that 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. A concatenated file can be accessed only for the duration of the logical name. You must specify the files in the concatenation order desired. Files can be specified 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 mark appears at the end-of-medium, the end-of-file is masked. This allows concatenated files to be accessed logically as a single file without the hindrance of intermediate end-of-file marks being returned. 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 priviledges 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 FILE SECURITY

In a DNOS system that has been generated with the file security option, there are two factors that affect how you can access a file. These factors are the access groups to which you belong and the access rights for those groups for any particular file you wish to use. The *DNOS Security Manager's Guide* describes how to set up a secure environment. In most cases, your security manager will determine what access groups exist in your environment and will assign you to one or more access groups. The security manager or some other access group leader may also be responsible for determining which files have what access rights for particular groups.

The commands for creating access groups and allowing various groups to access particular files can be available to you or they can be restricted to the security manager or some select group of users. The access rights to the command procedures, in addition to their privilege level, determine who can use which commands.

While using the commands, if you have file security, you will need the appropriate access to files you manipulate with the commands. The access rights available are read, write, delete, execute, and control.

In general, the read access right is needed for a file accessed by a command if that command shows data in the file or examines the file for input. For example, the Show File (SF) command requires that you have read access to the file being shown. If you do not have read access, you will receive an error message from SF.

The write access right is needed for a file accessed by a command that modifies or updates a file. For example, the Append File (AF) command requires access to the file used for OUTPUT PATHNAME. AF also requires read access to the files used as INPUT ACCESS NAME(S).

If you issue a command that deletes a file, you must have the delete access right to that file. Delete File (DF), for example, requires that you have delete access to the file(s) specified for PATHNAME(S). Since the text editor replaces an existing file with a new one, you need delete access to the file specified for FILE ACCESS NAME if you are replacing that file when using the Execute Text Editor (XE) and Quit Editor (QE) commands.

A command that executes a task from a program file requires that you have the execute access right for that program file. The Execute Task (XT) command, for example, requires that you have execute access to the file specified for PROGRAM FILE OR LUNO.

The control access right is required for any command that changes the access rights to a file. If you want to use the Modify Security Access Rights (MSAR) command, for example, you must have the control access rights to the file specified for FILE NAME.

The *DNOS SCI Reference Manual* describes the security commands. It also points out unexpected security implications for various SCI commands.

2.9 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).

2.9.1 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.

2.9.1.1 Resource-Specific I/O. Resource-specific I/O operations assume device, channel, or file peculiarities. For example, activating the graphic capability on a 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

2.9.1.2 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.

2.9.2 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. Requestor tasks (slaves) have less flexibility and fewer privileges.

2.9.2.1 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.

2.9.2.2 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 requester(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.

2.9.2.3 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 requestor 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.

2.9.2.4 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)

2.9.2.5 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.

2.9.3 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. TIP programs can use either the provided file-handling routines or execute SVCs directly via procedure SVC$.

2.9.4 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 hexadecimal 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.

2.9.5 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 ANSI 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 a spooler 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
- Two 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.

2.10 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.

You build the program, specify appropriate division of the program to the Link Editor, and install 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.

2.11 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.

2.11.1 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 Command 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 online expanded error messages or refer to the *DNOS Messages and Codes Reference Manual*.

2.11.2 Online Expanded Error Message Documentation

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

2.11.2.1 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:

```
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.
```

2.11.2.2 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.

2.11.3 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 Command key or Return key so that operation can continue. Expanded status messages can be obtained in the same way as error messages.

3

Building a TI Pascal Program

3.1 GENERAL

This section begins a step-by-step examination of TIP program development. Topics discussed include how to prepare the necessary directories and files and how to enter the program source code. The discussions reflect the assumption that the program source code will be entered via the Text Editor.

3.2 DIRECTORY AND FILE PREPARATION

During the development of a program, many different files are used. Some are permanent files that should be organized logically. Since persons other than the original programmer usually maintain the programs, the files should be organized in a way that facilitates finding and maintaining all related files.

3.2.1 Required Files

Table 3-1 shows the required files for developing and executing programs. Optional procedures may require additional files. These files are discussed within the context of each procedure.

Table 3-1 Files Required for Program Development

File	How and Where Used
Source file	Contains program source code, output from the Text Editor, input to the Nester (optional), and input to the TIP compiler if nested source is not used.
Nested source file	This file is recommended but not required. It contains program source code formatted to emphasize TIP block structure and is output from the Nester utility. It may be input to the TIP compiler.
Object file	Contains program object code, output from the TIP compiler, and input to the Link Editor.
Compiler listing file	Contains the program source listing with any errors detected by the TIP compiler, and additional information from the TIP compiler.
Link control file	Contains instructions for the Link Editor, such as the object file to be used for input, run-time or user libraries to be linked, external routines to be linked, and the program file to be output.
Link Editor listing (link map file)	Contains the listing of the link map output from the Link Editor.
Program file	The executable program file; contains linked object code output from the Link Editor.

3.2.2 Alternate Directory Structures

Each programmer or installation can decide how best to organize the files shown in Table 3-1. When a team of programmers is involved, a convention might be established.

Two possible methods of organization for the files are organization according to the related programs and organization according to file types. These two alternatives are illustrated in the following paragraphs and are only two of many possible structures.

3.2.2.1 Directories Organized by Programs. The basis for this method of organization is that all necessary files for a program are located in a single directory. This is illustrated by the following:

VOLUME.PROG1.SOURCE	VOLUME.PROG2.SOURCE
VOLUME.PROG1.OBJECT	VOLUME.PROG2.OBJECT
VOLUME.PROG1.LISTING	VOLUME.PROG2.LISTING
VOLUME.PROG1.LCONTROL	VOLUME.PROG2.LCONTROL
VOLUME.PROG1.LINKMAP	VOLUME.PROG2.LINKMAP
VOLUME.PROG1.PROGRAM1	VOLUME.PROG2.PROGRAM2

The directory name reflects the program to which all the files are related. All files for PROGRAM1 are contained in directory PROG1. Files for PROGRAM2 are in directory PROG2.

3.2.2.2 Directories Organized by File Type. In the following case, a directory is created for each file type. Using this structure, all source files are in one directory, object files are in another, and so on.

VOLUME.SOURCE.PROG1	VOLUME.SOURCE.PROG2
VOLUME.OBJECT.PROG1	VOLUME.OBJECT.PROG2
VOLUME.LISTING.PROG1	VOLUME.LISTING.PROG2
VOLUME.CONTROL.PROG1	VOLUME.CONTROL.PROG2
VOLUME.LINKMAP.PROG1	VOLUME.LINKMAP.PROG2
VOLUME.PROGRAM.PROG1	VOLUME.PROGRAM.PROG2

In this structure, the source files for both PROG1 and PROG2 are in the same directory, VOLUME.SOURCE. The same is true for all files of a given type.

3.2.3 Creating the Directories and Files

To create a directory or subdirectory, use the Create Directory File (CFDIR) command. Specify the directory pathname and the maximum number of entries (files or subdirectories) the directory is to contain. You may specify the default physical record size, or accept the default value established either during system generation or when the volume in use was initialized. The default physical record size is set to optimize file management and is usually the recommended size. However, if you intend to transport files from DNOS to another system, characteristics of the target system may dictate a physical record size other than the default.

To create the directory VOLUME.PROG1, which will contain all files used in developing PROGRAM1, enter CFDIR. The following display appears:

```
[] CFDIR
   CREATE DIRECTORY FILE
                        PATHNAME:  pathname@
                     MAX ENTRIES:  integer
    DEFAULT PHYSICAL RECORD SIZE:  [integer]
```

Assuming the name of the volume is VOLUME, the pathname is VOLUME.PROG1. Since a minimum of six files are required in the example, specify 12 as the maximum number of entries. Accept the system (or volume) default physical record size by pressing Return.

Files that are output from a utility (such as the Text Editor or compiler) need not be created prior to executing the utility. The pathnames must be specified, but the files are created automatically by the utility if they do not already exist. Directories are not automatically created. For example, the object file pathname is specified when the compiler is initiated. The directory must exist at that time, but the compiler automatically creates the file if the file does not already exist. The link control file is not a utility output file; consequently, it must be created (usually via the Text Editor) prior to executing the Link Editor.

A pathname may include several levels of directories, each created by the CFDIR command. However, all file names in a given directory must be unique.

3.2.4 Building a Program Using the Text Editor

A TIP program is usually created on VDT's using the Text Editor. File editing with a VDT occurs on page basis. A page is any 24 lines that appear on the VDT screen at one time. You can edit any record displayed on the screen by positioning the cursor within the line containing the record. You can insert records between lines and can insert or delete them in any order. Also, you can insert, delete, or modify characters within a line. To adjust the portion of the file being displayed, use the Show Line (SL) SCI command and the F1, F2, Previous Line, and Next Line keys. For further information about the Text Editor, refer to the *DNOS Text Editor Reference Manual.*

3.2.5 Example of Using the Text Editor

After you are properly logged-on and the necessary directories have been created, you are ready to enter the program. To activate the Text Editor, use the Initiate Text Editor (XE) SCI command. To do so, enter XE and press Return. The following display appears:

```
[] XE
EXECUTE TEXT EDITOR
  FILE ACCESS NAME:  filename@  (*)
```

Because there is no input file to be edited at this time, fill the field with spaces and press Return. (When responding to command prompts, press the Skip key once to blank the field from the cursor position to the end of the field. The cursor then moves to the beginning of the next field.)

The end-of-file (EOF) record, the only record in the file at this time, is displayed in the home position (the upper left corner):

```
*EOF
```

The Execute Text Editor with Scaling (XES) command is equivalent to the XE command with the addition of a scale on line 24. The scale, which represents column numbers, is useful when entering programs or data according to a structured format.

The Text Editor is initially in compose mode. When in compose mode, you can move the cursor to the beginning of a new blank line by pressing Return. This allows new lines to be added as the file is being built. By pressing Return now, the first line is created. When the Uppercase Lock key is pressed, letters are entered as capitals. The Uppercase Lock key affects only the letter keys.

To set tab stops within the Text Editor, use the Modify Tabs (MT) command. First, enter the command mode by pressing the Command key. The SCI prompt [ ] appears on line 24, indicating readiness to accept a command. Enter MT and press Return. The following display appears:

```
[] MT
MODIFY TABS
        TAB COLUMNS:  (column,...,column)
```

Enter the column numbers where tab stops are to be set, separating column numbers with commas. Press the Return key. The Text Editor file reappears with the cursor in the same position as before the MT command was entered. Move the cursor forward from tab stop to tab stop by using the Next Field key. Move the cursor backward by using the Previous Field key.

For entering or editing TIP source, use the Modify Right Margin (MRM) command to set the right margin at column 72. This allows the Nester utility to insert sequence numbers in columns 73 through 80. To change the margin, first press the Command key. Then, enter MRM and press Return. The following display appears:

```
[] MRM
MODIFY RIGHT MARGIN
   RIGHT MARGIN POSITION:  (80)
```

Enter 72 as the right margin position and press Return. The Text Editor file reappears with the cursor in the same position as before the MRM command was entered.

Use the procedures given in this section to enter the program shown in Figure 3-1.

```
(*$MAP,NO OPTIMIZE,WIDELIST,CKINDEX,CKSUB*)
PROGRAM DIGIO;
TYPE CHBUF = ARRAY (.1..6.) OF CHAR;
VAR BUFF      : CHBUF;
I,NUM      :INTEGER;
PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
BEGIN     (* CCHAR *)
NUM:=0;
FOR J:= 1 TO I DO
IF BUFF(.J.)>='0' AND BUFF(.J.) <='9'
THEN  NUM:= NUM*10+ORD(BUFF(.J.))-ORD('0')
END;

PROCEDURE CINT (NUM:INTEGER);
VAR I          :INTEGER;
BEGIN     (* CINT *)
I:= NUM DIV 10;
IF I <> 0 THEN CINT(I);
WRITE (CHR(NUM MOD 10 + ORD('0')))
END;

BEGIN (* DIGIO *)
WRITELN('ENTER 1 TO 5 DIGITS');
RESET(INPUT);
I:=I+1;
WHILE NOT EOLN DO BEGIN  (* INPUT CHARS *)
READ (BUFF(.I.));
I:= I+1;
END;  (* INPUT CHARS *)
I := I-1 ;
CCHAR(BUFF,NUM,I);
NUM:=NUM+25;
CINT(NUM);
WRITELN;
END.
```

Figure 3-1 Example Program: DIGIO

For new programs, it is recommended that you turn on the compiler options MAP, WIDELIST, and the check options (CKINDEX, CKSUB, etc.) during the debugging phase of program development. (Compiler options are described in the *TI Pascal Reference Manual*.) The MAP option provides variable locations that are needed for interpreting memory dumps. The WIDELIST option provides line numbers, which are referenced by some of the run-time error messages. The check options provide additional run-time error checking that can help you locate mistakes in the program. When you have checked out the program, recompile it with the check options turned off to reduce the size of the object code.

After entering the program source, use the F2 key to roll the file back to the beginning of the file. Review the source code for errors. To edit the file, press the F7 key to switch the Text Editor to edit mode. Pressing Return in edit mode moves the cursor to the beginning of the next line without creating a new blank line. (In either edit or compose mode, a blank line can be inserted above the cursor by pressing the Initialize Input key.)

When the file has been entered and edited, use the Quit Edit (QE) command to terminate the Text Editor. Press the Command key to enter the command mode, enter QE, and press Return. The following display appears:

```
[] QE
QUIT EDIT
     ABORT?:  {YES/NO}  (NO)
```

The reply to the ABORT? prompt determines if the data entered is to be retained (NO response) or discarded (YES response). Since the source code is to be retained, press Return to accept the default value (NO). The following display appears:

```
QUIT EDIT
     OUTPUT FILE ACCESS NAME:   pathname@ (*)
                    REPLACE?:   {YES/NO} (NO)
        MOD LIST ACCESS NAME:   [pathname@]
```

Enter the pathname of the output file (for example VOLUME.PROG1.SOURCE). Since this is a new file being created, press Return to accept the default value (NO) to the REPLACE? prompt. When the edited output file is to replace a file of the same name, the response to the REPLACE? prompt is YES. Skip through the last prompt by pressing Return. (See the *DNOS Text Editor Reference Manual* for a discussion of the MOD LIST ACCESS NAME.) The Text Editor output file is written, and the main menu appears. If at another time the file VOLUME.PROG1.SOURCE is to be edited, enter that pathname as the input file access name for the Text Editor.

4

Nester Utility

4.1 GENERAL

This section includes guidelines to assist in coding a source program and describes the Nester utility. Nester provides a standardized block format for source code, performs rapid syntax checking, and helps format source code for input to the configuration processor (Section 6).

Typically, the TIP source program is written on a disk file using the Text Editor as described in Section 3. For a program with 80-column records, Nester may be used with the WIDE option to convert from 80-column to 72-column format.

4.2 NESTER FUNCTIONS

Nester uses five parameters in establishing the source program format that corresponds to the logical organization of the program. Table 4-1 lists these parameters and their default values, along with four other options. Two of these parameters, DTAB and CCOL, apply to the declarations of the program. The rules that apply to each of the declarations are as follows:

- The PROGRAM declaration begins in the leftmost character position (column 1). Example:

  ```
  PROGRAM DIGIO;
  ```

- The LABEL declaration begins in the leftmost character position (column 1), and the label numbers follow on the same line. Example:

  ```
  LABEL 10,20,30;
  ```

- The CONST declaration begins in the leftmost character position (column 1), and constants are tabulated into columns separated by the value of the Constant Column parameter CCOL. Example:

  ```
  CONST A = 2;          B = 3;
    C = 4;              D = 5;
    E = 6;              F = 7;
  ```

- The TYPE declaration begins in the leftmost character position (column 1). The first type declaration follows on the same line. Example:

  ```
  TYPE A = ARRAY(.0..10.) OF INTEGER;
  ```

- Any subsequent type declarations are indented by the value of the Declaration Tab parameter DTAB. Example:

```
TYPE A = ARRAY(.0..10.) OF INTEGER;
     B = (C,D,E);
```

- In a record declaration, the first field identifier follows the declaration; subsequent field identifiers are indented on a new line by the value of DTAB. Example:

```
TYPE A = ARRAY(.0..10.) OF INTEGER;
  B = (C,D,E);
  REC = RECORD A:INTEGER;
    B:REAL;
    CASE C:BOOLEAN OF
      FALSE:(D:INTEGER;
        E:INTEGER);
    END;
```

- When the component type of an array is a structured type, the structured type is indented on a new line by the value of DTAB. Example:

```
TYPE A = ARRAY(.0..10.) OF INTEGER;
  B = (C,D,E);
  REC = RECORD A:INTEGER;
    B:REAL;
    CASE C:BOOLEAN OF
      FALSE:(D:INTEGER;
        E:INTEGER);
    END;
  MATRIX = ARRAY(.0..9.) OF
    ARRAY(.0..9.) OF
      ARRAY(.0..2.) OF INTEGER;
```

- The VAR declaration begins in the leftmost character position (column 1). The first variable declaration follows on the same line. Subsequent variable declarations are indented by the value of DTAB, and each starts on a new line. Example:

```
VAR ARY : A;
  I,J,K,L:INTEGER;
  R:REC;
```

- The COMMON declaration begins in the leftmost character position (column 1). The common variables follow on the same line. Example:

```
COMMON R:BOOLEAN;
  I:INTEGER;
  X:REAL;
  M:ARRAY(.1..5.) OF INTEGER;
```

- The ACCESS declaration begins in the leftmost character position (column 1). The common variables follow on the same line. Example:

  ```
  ACCESS COM1,COM2;
  ```

- The routine declarations begin with the keyword PROCEDURE or FUNCTION in the leftmost character position (column 1). The parameter list follows on the same line. Example:

  ```
  PROCEDURE A(B,C:INTEGER);
  ```

Table 4-1. Nester Options

Option Keyword	Meaning	Default
DTAB	Declaration tab value.	4
CCOL	Constant column increment.	20
STAB	Statement tab value.	2
SCOL	Statement column increment.	1
SLIM	Statement column limit.	30
ADJT	Adjust comments to right margin.	TRUE
FILL	Allows concatenation of source lines in accordance with options in effect. When FALSE, items on a new line of input are placed on a new line in the output.	TRUE
CONV	Convert characters in source statements to those of a transportable character set.	FALSE
WIDE	Accept 80-column source statements.	FALSE
NUMB	Place line numbers in columns 73 – 80 when TRUE.	TRUE

In addition to the rules for the declarations, another rule applies to breaking lines in the declaration section. A line starts with the declaration keyword or an identifier and is usually separated from the next line by a semicolon (and the end-of-line indication appropriate to the input device). When Nester breaks a line, the continuation on the succeeding line is indented by the value of DTAB.

Two other parameters, STAB and SCOL, apply specifically to the statement portions of the program, that is, the compound statements that contain the statements that specify the processing of the program. The rules that apply to the statement portions are as follows:

- The BEGIN and END statements of the compound statement that contains the statements of the block begin in the leftmost character position (column 1).
- The component statements within the compound statement are indented by the value of STAB.
- A statement is placed on a new line if it is too long to fit on the current line.
- When a statement is too long for a single line, the continuation line is indented by the value of STAB.
- Structured statements other than compound statements (IF, CASE, FOR, WHILE, REPEAT, and WITH) always begin on a new line.
- Component statements within a structured statement are indented by the value of STAB.
- Keywords BEGIN and END do not cause indentation and normally do not begin a new line.
- Statement labels and escape labels start in character position two and may force the labeled statement to start on a new line.
- The ELSE portion of an IF statement starts a new line and is positioned at the same character position as the corresponding IF.
- The UNTIL portion of a REPEAT statement starts a new line and is positioned at the same character position as the corresponding REPEAT.
- Each CASE label is indented by the value of STAB.
- Each CASE alternative statement (following keyword OTHERWISE) is indented by the value of STAB beyond the character position at which the keyword OTHERWISE begins.

Option ADJT enables or disables the adjustment of comments. When the value of ADJT is true (default), comments that are less than 70 characters long are right justified on the line on which they are entered. Comments that are more than 69 characters long are positioned to begin in character position one. Comments that cross line boundaries are not moved. When the value of ADJT is false, comments are not moved.

The column limit value SLIM applies to both declarations and statements that are arranged in columns. Specifically, these are constants and simple statements. No constant or statement starts beyond the column limit. For example, the default value of 30 allows only two columns of constants in a CONST declaration and only as many simple statements as can be placed on a line without starting a statement beyond character position 30.

The FILL option enables or disables the concatenation of characters from two or more lines to one line. When the value of the option is true (default), constants in a CONST declaration may be arranged in columns and simple statements may be concatenated on one line, using the value of the CCOL and SCOL options, respectively. When the value of the option is false, constants and simple statements remain on the line on which they are entered.

The CONV option converts the characters in source statements that are not transportable to characters that are more generally available. Specifically, lowercase letters are converted to uppercase letters, brackets ("[" and "]") are replaced with the equivalent character combinations ("(." and ".)"), and braces ("{" and "}") are replaced with the equivalent character combinations ("(*" and "*)").

The WIDE option accepts 80-column source statements and reformats the source code to 72-column output. When the input source statement contains a string constant longer than 72 characters, Nester issues an error message.

The NUMB option places line numbers in columns 73 through 80 when true (default). Setting NUMB to false suppresses line numbers, which speeds printing.

Figure 4-1 and Figure 4-2 show the effect of Nester on example program DIGIO (from Section 3), using the default options.

```
(*$MAP,NO OPTIMIZE,WIDELIST,CKINDEX,CKSUB*)
PROGRAM DIGIO;
TYPE CHBUF = ARRAY(.1..6.) OF CHAR;
VAR BUFF      : CHBUF;
I,NUM      :INTEGER;
PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
BEGIN      (* CCHAR *)
NUM:=0;
FOR J:= 1 TO I DO
IF BUFF(.J.)>='0' AND BUFF(.J.)<=' 9'
THEN NUM:= NUM*10+ORD(BUFF(.J.))-ORD('0')
END;

PROCEDURE CINT (NUM:INTEGER);
VAR I          :INTEGER;
BEGIN      (* CINT *)
I:= NUM DIV 10;
IF I <> 0 THEN CINT(I);
WRITE (CHR(NUM MOD 10 + ORD('0')))
END;

BEGIN (* DIGIO *)
WRITELN('ENTER 1 TO 5 DIGITS');
RESET(INPUT);
I:=I+1;
WHILE NOT EOLN DO BEGIN (* INPUT CHARS *)
READ (BUFF(.I.));
I:= I+1;
END;    (* INPUT CHARS *)
I := I-1;
CCHAR(BUFF,NUM,I);
NUM:=NUM+25;
CINT(NUM);
WRITELN;
END.
```

Figure 4-1. Source Code as Input to Nester

```
              (*$MAP,NO OPTIMIZE,WIDELIST,CKINDEX,CKSUB*)                    00000010
PROGRAM DIGIO;                                                               00000020
TYPE                                                                         00000030
    CHBUF = ARRAY(.1..6.) OF CHAR;                                           00000040
VAR BUFF : CHBUF;                                                            00000050
    I,NUM :INTEGER;                                                          00000060
PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);                      00000070
BEGIN                                                        (* CCHAR *)    00000080
  NUM:=0;                                                                    00000090
  FOR J:= 1 TO I DO                                                          00000100
    IF BUFF(.J.)>='0' AND BUFF(.J.)<='9' THEN                                00000110
      NUM:=NUM*10+ORD(BUFF(.J.))-ORD('0')                                    00000120
END;                                                                         00000130
                                                                             00000140
PROCEDURE CINT (NUM:INTEGER);                                                00000150
VAR I :INTEGER;                                                              00000160
BEGIN                                                         (* CINT *)    00000170
  I:= NUM DIV 10;                                                            00000180
  IF I <> 0 THEN CINT(I)                                                     00000190
  WRITE (CHR(NUM MOD 10 + ORD('0')))                                         00000200
END;                                                                         00000210
                                                                             00000220
BEGIN                                                         (* DIGIO *)   00000230
  WRITELN('ENTER 1 TO 5 DIGITS');                                            00000240
  RESET(INPUT);                                                              00000250
  I:=I+1;                                                                    00000260
  WHILE NOT EOLN DO BEGIN                                  (* INPUT CHARS *) 00000270
    READ (BUFF(.I.));                                                        00000280
    I:= I+1;                                                                 00000290
    END;                                                   (* INPUT CHARS *) 00000300
  I := I-1;                                                                  00000310
  CCHAR(BUFF,NUM,I);                                                         00000320
  NUM:=NUM+25;                                                               00000330
  CINT(NUM);                                                                 00000340
  WRITELN;                                                                   00000350
END.                                                                         00000360
```

Figure 4-2. Nested Source Code, Output From Nester

4.3 NESTER OPTION COMMENT

The values of the Nester options are changed by entering option comments. The value supplied in an option comment continues to apply until another value is supplied in a subsequent option comment. The option comments become a part of the source program and are processed by the compiler as comments. That is, they have no effect on compiler options. The syntax of an option comment is as follows:

<option comment> ::= "{"&<option> [,<option>]"}"

<option> ::= <integer-valued option> (integer constant)
| <Boolean-valued option> <plus or minus>

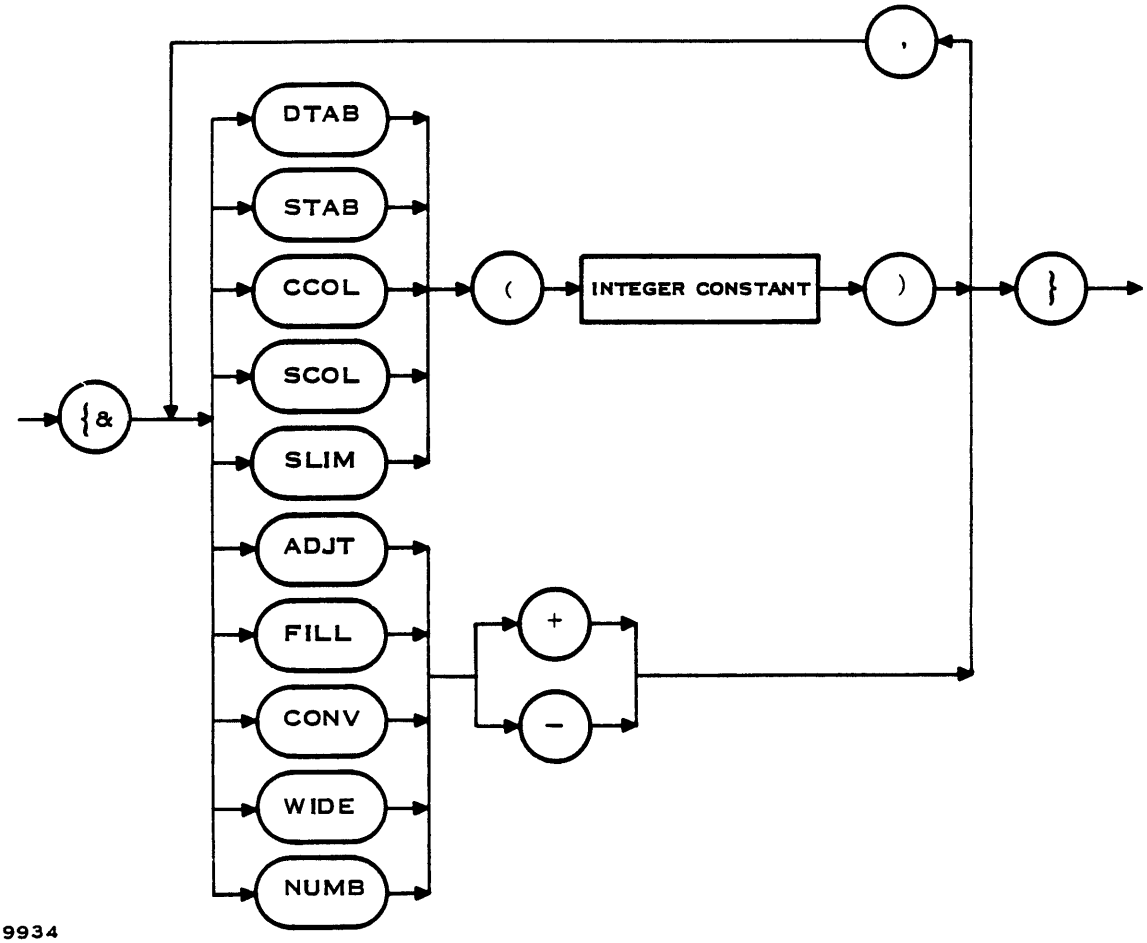
<integer-valued option> ::= DTAB|STAB|CCOL|SCOL|SLIM

<Boolean-valued option> ::= ADJT|FILL|CONV|WIDE|NUMB

<plus or minus> ::= + | -

The syntax diagram is as follows:

Option comment:



2279934

No blanks are allowed in the option comment. The option names may be entered in any sequence.

EXAMPLES

{&DTAB(2),STAB(5),FILL-,SLIM(72)}

{&DTAB(4),CCOL(20),STAB(2),SCOL(1),SLIM(30),ADJT+,FILL+}

The first example changes the value of DTAB to 2 and the value of STAB to 5. It sets the value of the FILL option to false, and changes the value of SLIM to 72. This value for SLIM allows maximum packing of constants on a single line and maximum packing of simple statements on a single line. However, as a result of setting FILL to false, the maximum packing allowed by setting the value of SLIM to 72 does not occur.

The second example restores the option values to the default values.

4.4 EXECUTING NESTER

TIP software includes an SCI procedure XNESTER for executing the source formatter, Nester. Enter XNESTER any time the system is ready for a command. The following command prompts appear:

```
[] XNESTER
EXECUTE PASCAL SRC PROGRAM NESTER  <VERSION X.X.X YYDDD>

                SOURCE: pathname@
          NESTED SOURCE: pathname@
         NESTER OPTIONS: [option,...,option] (SLIM (0))
          ERROR LISTING: pathname@ (ME)
              MESSAGES: [pathname@] (ME)
                  MODE: {FOREGROUND/BACKGROUND} (FOREGROUND)
```

The prompts for NESTER are as follows:

- SOURCE — The access name of a file that contains source code to be formatted
- NESTED SOURCE — The access name of a file for the formatted source code
- NESTER OPTIONS — The NESTER options to be in effect (see Table 4-1). The option list follows the same syntax as for <options> described in paragraph 4.3, with the exception that the option list should not be enclosed in braces.
- ERROR LISTING — The access name of a device for listing syntax errors
- MESSAGES — The access name of a device or file for system messages

The mode can be either foreground or background. When Nester is executing in the background mode, the terminal is available for entry of other commands or for foreground execution of another program. When Nester is executed in the foreground mode (default), the terminal may not be used by any program other than Nester until Nester terminates execution. Nester displays the following termination message:

```
EXECUTION OF SOURCE PROGRAM NESTER IS COMPLETE:
```

Nester uses four files, as follows:

File Name	Contents
OUTPUT	Commands executed and error messages
SYSMSG	System messages
NESTSRC	Source module to be reformatted
NESTOUT	Reformatted source module

4.5 NESTER ERROR MESSAGES

Nester checks the statements in the source file for simple syntax errors and prints error numbers when errors are found. The nested source code that results from nesting source statements that contain these errors is usually incorrectly nested. For this reason, the source file and the nested source file cannot be the same.

Table 4-2 lists the Nester error messages and corresponding error numbers. Generally, the Nester error numbers correspond to those for comparable compiler errors.

Table 4-2. Nester Errors

Number	Message
2	Identifier expected
4	) expected
5	: expected
6	Illegal symbol
8	OF expected
9	(expected
10	Error in type
11	[expected
12	] expected
13	END expected
14	; expected
15	Integer expected
16	= expected
17	BEGIN expected
50	Error in constant
51	:= expected
52	THEN expected
53	UNTIL expected
54	DO expected
55	TO/DOWNTO expected
58	Error in factor
106	Number expected
201	Error in REAL constant; digit expected
202	String constant too long or crosses a source line boundary
255	Too many errors on this source line

5

TIP Compiler

5.1 GENERAL

The TIP compiler consists of six separate tasks:

- PREPROC — Preprocesses the source file to provide conditional compilation and copy file processing
- SILT1 — Source-to-intermediate-language-translator
- SILT2 — Source-to-intermediate-language-translator
- T9OPT — Performs optimization on the intermediate language
- CODEGEN — Translates the intermediate language into object code
- PSCLXREF — Provides a cross-reference listing of the TIP source

This section describes the following: default pathnames for files used by TIP tasks, the compiler's general flow of execution and the files used during a compilation, the source management directives supported by the processor task, the system procedures needed to invoke the compiler, the compiler listing, and the compiler message file contents.

5.2 COMPILER EXECUTION OVERVIEW

This overview begins with a discussion of the default pathnames for files used by TIP tasks, then describes the compiler's flow of execution and the files used by each compiler task (Table 5-1).

5.2.1 Default Pathnames

Default pathnames are assigned for input, output, object, and message files when you do not specify pathnames for these files. The default pathnames are derived from the logical files names INPUT, OUTPUT, OBJECT, and SYSMSG. The system appends the numeric characters of the initiating station's ID to the logical file name. If, for example, you are executing the TIP compiler from ST04 and do not specify pathnames for the object or listing files, the default pathnames would be .OBJECT04 and .OUTPUT04, respectively. If you do not specify a message file, its default pathname is .SYSMSG04. Some procedures will accept a default input file, such as .INPUT04, but the file must exist and contain the desired data for the procedure to execute properly. The default pathnames begin with a period, indicating that they reside on the system disk.

When the compiler is executed from a batch stream or batch job the name of a temporary directory is prefixed to the default pathname. Synonym $TIP is automatically assigned to this directory. For batch streams initiated by the Execute Batch (XB) command, the temporary directory is .T$STxx where xx is the station number of the initiating terminal. For batch jobs initiated by the Execute Batch Job (XBJ) command, the directory is .T$Jn where n is the job number (from one to five decimal digits long). In both cases, the temporary directory is automatically deleted when the batch stream ends. To place the files elsewhere, assign synonym $TIP to a directory you have created. However, you must delete the files yourself when finished with them.

5.2.2 Preprocessor

The first task of the TIP compiler is the preprocessor task, PREPROC. At your option, this task implements conditional compilation and copy statements. It has one input and two output files. The file INPUT contains the user source file to be compiled. PREPROC creates the listing file OUTPUT and the file PREOUT, which is the source file after it has been preprocessed. It also writes information to the SYSMSG file.

5.2.3 SILT1

SILT1 parses the source statements and translates them into an intermediate (token) representation that is used as input to SILT2. SILT1 reads the source program from file PREOUT and writes the intermediate representation on file TOKENS. SILT1 detects syntax errors and writes information about detected errors on file ERRFILE to be included in the listing written by SILT2. The character strings for each identifier are written to IDFILE, and the listing page header information is written to LISTFILE. SILT1 also writes the first eight characters of the name of each routine on file SYSMSG for display.

Table 5-1. Files Required by the Compiler Tasks

Task	File Name	I/O	Contents
PREPROC	INPUT	I	TIP source program
	PREOUT	O	Preprocessed source
	SYSMSG	O	Messages
SILT1	PREOUT	I	Preprocessed source
	OUTPUT	O	Used as dump file
	SYSMSG	O	Messages
	*TOKENS	O	Parse tokens (input to SILT2)
	*ERRFILE	O	Syntax errors (input to SILT2)
	*IDFILE	O	Identifiers (input to SILT2)
	*LISTFILE	O	Page header information
SILT2	PREOUT	I	Preprocessed source
	OUTPUT	O	Source listing with errors

Table 5-1. Files Required by the Compiler Tasks (Continued)

Task	File Name	I/O	Contents
	SYSMSG	O	Messages
	*TOKENS	I	Parse tokens
	*ERRFILE	I	Syntax errors
	*CILFIL	O	Common intermediate language
	*DESCFIL	O	Module descriptor file
	*IDFILE	I	Identifiers
	*LISTFILE	I/O	Page header information
	*TEMPFILE	I/O	Probe Information
T9OPT	OUTPUT	O	Optimization information
	SYSMSG	O	Messages
	*CILFIL	I	Common intermediate language
	*DESCFIL	I	Module descriptor file
	*LISTFILE	I/O	Page header information
	*CIL$FIL	O	Common intermediate language
	*DE$CFIL	O	Module descriptor file
CODEGEN	OBJECT	O	Object file
	OUTPUT	O	CODEGEN summary information
	SYSMSG	O	Messages
	*CILFIL	I	Common intermediate language
	*DESCFIL	I	Module descriptor file
	*TMPFIL	I/O	Intermediate Object
	*LISTFILE	I	Page header information

Table 5-1. Files Required by the Compiler Tasks (Continued)

Task	File Name	I/O	Contents
PSCLXREF	PREOUT	I	Preprocessed source
	OUTPUT	O	Cross-reference listing
	SYSMSG	O	Messages

Note:

* This is an internal temporary file.

5.2.4 SILT2

SILT2 transforms the intermediate representation on file TOKENS into an intermediate language written on file CILFIL as input to T9OPT. SILT2 also writes a descriptor file, DESCFIL, for input to T9OPT. SILT2 detects semantic errors in the source program and writes a source listing. Errors that SILT1 writes to ERRFILE are combined with errors detected by SILT2 and are then written to file OUTPUT. Also written to file OUTPUT are the source code from file PREOUT, the listing page header information from file LISTFILE, and the variable map that uses the character strings of file IDFILE. SILT2 writes the first eight characters of the name of each routine on file SYSMSG and writes an error message when any errors have been detected by either SILT1 or SILT2.

5.2.5 T9OPT

T9OPT performs optimization functions on CILFIL, using the descriptions on DESCFIL. Optimization information and listing page header information from LISTFILE is written on file OUTPUT. The first eight characters of each module name are written on file SYSMSG.

5.2.6 CODEGEN

CODEGEN transforms the intermediate language on CILFIL into object code, using the descriptors on DESCFIL, and writes the object code to a sequential file, OBJECT. CODEGEN writes to an internal temporary file, TMPFIL, from which it retrieves the data when required. CODEGEN completes the listing on file OUTPUT by writing statistical information, such as the number of instructions generated and the storage requirements for each module. CODEGEN also writes the first eight characters of each module name on file SYSMSG.

5.2.7 Cross-Reference

PSCLXREF scans the PREOUT source file and produces a cross-reference listing, which is written to file OUTPUT. It also writes information to file SYSMSG.

5.3 COMPILER SCI COMMANDS

The following paragraphs describe the six commands that invoke phases of the TIP compiler: XTIP, XTIPL, XSILT, XCODE, XPP, XPX, and XALX. These command procedures prompt for certain file pathnames, some of which can be assigned by the system.

5.3.1 XTIP

The Execute TI Pascal Compiler (XTIP) command executes the entire compilation process. XTIP executes all six phases (with XREF optional) in the following order: PREPROC, SILT1, SILT2, T9OPT, CODEGEN, and PSCLXREF. The prompts are as follows:

```
[] XTIP
EXECUTE TI PASCAL COMPILER <VERSION: X.X.X YYDDD>


     SOURCE:  pathname@ (*)
     OBJECT:  [pathname@] (*)
    LISTING:  [pathname@] (*)
   MESSAGES:  [pathname@] (*)
    OPTIONS:  [option,...,option]
       MEM1:  [integer,integer] (6,4) (SILT1 phase)
       MEM2:  [integer,integer] (13,4) (SILT2 phase)
       MEM3:  [integer,integer] (10,4) (CODEGEN phase)
```

The MEM prompts specify the number of 1024-byte memory blocks to be allocated to the stack and heap for each compiler phase. The sum of the two numbers must not exceed 35 for MEM1, 21 for MEM2, and 20 for MEM3. XTIP displays the following termination message, along with a notice of any errors or warnings:

```
TIP COMPILATION COMPLETE:
```

5.3.2 XTIPL

The Execute TI Pascal Compile and Link (XTIPL) command executes the entire compilation process, like the XTIP command. It then builds a link control file to perform a simple task-only link edit, and executes the Link Editor. (Section 7 contains a discussion on link editing a TIP program.) The prompts for the XTIPL command are as follows:

```
[] XTIPL
EXECUTE TI PASCAL COMPILE AND LINK <VERSION: X.X.X YYDDD>


       SOURCE:  pathname@ (*)
       OBJECT:  [pathname@] (*)
      LISTING:  [pathname@] (*)
     MESSAGES:  [pathname@] (*)
      OPTIONS:  [option,...,option]
 PROGRAM FILE:  pathname@ (*)
    TASK NAME:  character(s) (*)
 LINK LISTING:  pathname@ (*)
```

Provided that the compiler terminates normally, XTIPL then executes the Link Editor using the following link control file:

```
FORMAT IMAGE, REPLACE
LIBRARY .TIP.OBJ
TASK <specified task name>
INCLUDE (MAIN)
INCLUDE <specified object file>
END
```

The linked output is installed on the specified program file, replacing an identically-named task if it exists. The TIP run-time library .TIP.OBJ applies to tasks using SCI for I/O access, so this link control file is not appropriate for tasks that use LUNO I/O.

5.3.3 XSILT

The Execute TIP Syntax Checker (XSILT) command executes all parts of the compiler except code generation. It is primarily used for syntax checking. The prompts for XSILT are the same as those for XTIP except for OBJECT and MEM3:

```
[] XSILT
EXECUTE TIP SYNTAX CHECKER <VERSION: X.X.X YYDDD>
      SOURCE:  pathname@ (*)
     LISTING:  [pathname@] (*)
    MESSAGES:  [pathname@] (*)
     OPTIONS:  [option,...,option]
        MEM1:  [integer,integer] (6,4) (SILT1 phase)
        MEM2:  [integer,integer] (13,4) (SILT2 phase)
```

5.3.4 XCODE

The Execute TI 990 CODEGEN (XCODE) command executes only the CODEGEN phase of the TIP compiler. It is normally used in conjunction with XSILT. The prompts are as follows:

```
[] XCODE
EXECUTE TI 990 CODEGEN <VERSION: X.X.X YYDDD>

      OBJECT:  [pathname@] (*)
     LISTING:  [pathname@] (*)
    MESSAGES:  [pathname@] (*)
      MEMORY:  [integer,integer] (10,4)
MODE(B,F,D):  {FOREGROUND/BACKGROUND/DEBUG}  (BACKGROUND)
```

5.3.5 XPP

The Execute Pascal Preprocessor (XPP) command executes only the preprocessor phase of the TIP compiler. The prompts are as follows:

```
[] XPP
EXECUTE PASCAL PREPROCESSOR  <VERSION: X.X.X YYDDD>

            INPUT:  pathname@ (*)
           OUTPUT:  [pathname@] (*)
          LISTING:  [pathname@]
         MESSAGES:  [pathname@] (*)
          OPTIONS:  [option,...,option]
```

5.3.6 XPX

The Execute Pascal Cross-Reference (XPX) command executes only the cross-reference phase of the TIP compiler. The prompts are as follows:

```
[] XPX
EXECUTE PASCAL CROSS-REFERENCE  <VERSION: X.X.X YYDDD>

              INPUT:  [pathname@]  (*)
             OUTPUT:  [pathname@]  (*)
           MESSAGES:  [pathname@]  (*)
NUMBER OF LINES/PAGES:  integer (60)
          MODE(F,B):  {FOREGROUND/BACKGROUND} (FOREGROUND)
```

5.3.7 XALX

TIP routines compiled with the LISTOBJ option produce an assembly language listing of the code generated by the compiler (refer to paragraph 5.5.4). The assembly language extractor (XALX) is a utility which extracts assembly code appearing in a compiler listing and changes it into a valid format for input to the macro assembler. The XALX command displays the following prompts:

```
[]XALX
EXECUTE TIP ASSEMBLY LANGUAGE EXTRACTOR <VERSION:X.X.X YYDD>
      INPUT LISTING ACCESS NAME:  pathname@     (*)
      OUTPUT SOURCE ACCESS NAME:  [pathname@]   (*) ACCESS NAME
                       MESSAGES:  [pathname@]   (*)
                      MODE(F,B):  {FOREGROUND/BACKGROUND} (FOREGROUND)
```

Specify a compiler listing file in response to INPUT LISTING ACCESS NAME. Specify a sequential file to the prompt OUTPUT SOURCE ACCESS NAME to receive the resulting assembly language source file.

5.3.8 P$DELETE

P$DELETE is an SCI command that deletes TIP temporary files.

```
[] P$DELETE
DELETE TIP TEMPORARY FILES  <VERSION: X.X.X YYDDD>
          STATION: station number (*)
```

5.3.9 P$SYN

P$SYN is an SCI command that deletes TIP synonyms, helping to alleviate or prevent synonym table overflow.

5.3.10 Options Prompt

This discussion of the OPTIONS prompt applies to the XTIP, XTIPL, XSILT, and XPP commands. The prompt OPTIONS is used to indicate execution mode, form of preprocessor output, number of lines per page, print width of a compiler line, whether to perform a cross-reference, and whether to include certain compiler options. Within the field, separate options with commas. In a batch stream, enclose the option list in double quotation marks ("option,option,...").

5.3.10.1 Mode of Execution. To indicate the compiler's mode of execution, enter FOREGROUND or BACKGROUND. F and B are sufficient abbreviations. The default mode is background. Entering ME for the MESSAGES prompt causes foreground execution regardless of this option.

5.3.10.2 Lines Per Page. To specify the number of lines per page of a listing, enter the phrase NUMBER OF LINES/PAGE or its abbreviation, NUM, followed by an equal sign and an integer. For example, NUM = 80 specifies that 80 lines per page be generated. The default is 60.

5.3.10.3 Print Width. To specify the print width, enter the phrase PRINT WIDTH or its abbreviation, PW, followed by an equal sign and an integer. For example, PW = 132 creates a file with a logical record length of 132 bytes. The default is 80.

5.3.10.4 Cross-Reference. To specify cross-reference, enter the word XREFERENCE or its abbreviation, X. The default is no cross-reference.

5.3.10.5 Disabling Source Preprocessing. To disable the preprocessing of the source by the preprocessor task, enter the phrase NO PREPROCESSOR or its abbreviation, NOPR. This option allows slightly faster compilation when no source management directives are used.

NOTE

If the preprocessing of source is disabled, the following options are not available.

5.3.10.6 Controlling Preprocessor Output. The form of preprocessor output (input to SILT1) is controlled by the phrase SUPPRESS PREPROCESSOR LINES. Normally, preprocessor commands and code in the false clause of a ?IF source management directive do not appear in the preprocessor output. (See the *TI Pascal Reference Manual* for details on source management directives.) To cause these lines to appear on the compiler listing, enter the option phrase SUPPRESS PREPROCESSOR LINES = NO or its abbreviation, SUP = N. This option is not available if NO PREPROCESSOR has been specified.

5.3.10.7 Compiler Options in the Procedure. The following compiler options can be specified either in the source code (as described in the *TI Pascal Reference Manual)* or in response to the OPTIONS prompt unless NO PREPROCESSOR is also specified:

NO ASSERTS	FORINDEX	NO LIST
NO WIDELIST	NO WARNINGS	SIDEFFECTS
NO GLOBALOPT	MAP	NO MAP
NO OPTIMIZE	PROBER	PROBES
NO TRACEBACK	UNSAFEOPT	WIDELIST
GLOBALOPT	SPEEDOPT	LOCALS
CKINDEX	CKOVER	CKPREC
CKSET	CKSUB	CKTAG
CKPTR	990/12	LISTOBJ

These compiler options are inserted into the source as the first line of the source file. The total number of characters that can be inserted into the source file is 46. The compiler options can be abbreviated according to the rules for abbreviating SCI parameters. (Generally, the abbreviation must contain enough characters to identify the option uniquely; refer to the *DNOS System Command Interpreter (SCI) Reference Manual.*) For instance, NOA is a sufficient abbreviation for NO ASSERTS. The 46-character limit applies to the unabbreviated form of the compiler options, which must be separated by commas.

The following options list specifies cross-reference, 80 lines to a page, displayed preprocessor commands, foreground execution, and TIP compiler options WIDELIST, UNSAFEOPT, and SIDEFFECTS. You can specify the options in any order.

```
OPTIONS: X,NUM = 80,WIDE,SUP = NO,F,UNSAFEOPT,SIDEFFECTS
```

5.4 ERROR HANDLING

Each task of the compiler sets a condition code, assigning the code as the value of synonym $$CC. The code is zero for normal termination and a nonzero value for abnormal termination. Specifically, values of $$CC following execution of the compiler have the following significance:

Value	Meaning
>0000	No errors or warnings
>4000	Warnings issued
>6000	Nonfatal errors detected
>8000	Fatal errors detected
>C000	Abnormal termination — compiler terminated with run-time error

NOTE

The angle bracket, >, preceding a number indicates a hexadecimal value.

Condition code synonym $$CC may be tested in a batch stream to alter the execution of the batch stream when an error is detected. The value placed in $$CC is valid only after the execution of the compiler (prior to execution of DNOS utilities that set $$CC). Its value may be stored in another synonym by setting that synonym to the value of @ $$CC with an .SYN SCI primitive. An attempt to determine the value of $$CC by executing a List Synonyms (LS) command always shows the value of $$CC as zero because the LS command sets $$CC to zero.

The compiler categorizes errors as warnings, nonfatal errors, and fatal errors. A warning is an irregularity that may or may not be an error. A nonfatal error is a syntax error that SILT corrects; it does not prevent CODEGEN from writing an object module or modules for the program. A fatal error is one that causes SILT2 to produce incorrect intermediate language; if CODEGEN executes, the resulting object module issues a run-time error when it is executed.

Appendix B lists and identifies the TIP error messages.

An abnormal termination occurs when a run time or internal error prevents the task from processing completely. Examples include an invalid file pathname or an insufficient amount of memory specified. These errors are reported in the MESSAGES file and on the user's terminal or batch stream listing. Appendix B lists the run-time error messages.

5.5 COMPILER LISTING

The TIP compiler listing is divided into five parts: preprocessor summary, source listing with errors, optimization summary with symbol map, CODEGEN summary, and cross-reference.

5.5.1 Preprocessor Summary

The preprocessor summary consists of echoed procedure parameters, the copy command's pathnames, and preprocessor error messages. Figure 5-1 shows a sample preprocessor summary:

```
PREPROC            1.8.0        83.357     PASCAL PREPROCESSOR

SOURCE = .SOURCE.IN1
OBJECT = .OBJECT04
LISTING = .SOURCE.LIST
MESSAGE = .SOURCE.MSG
MEM1 = 6,10
MEM2 = 13,4
MEM3 = 10,8
PRINT WIDTH = 80
NUMBER OF LINES/PAGE = 60
OPTIONS = (*$WIDELIST,MAP*)
SUPPRESS PREPROCESSOR LINES = YES

LINE NUMBER              COPY FILE PATHNAME
          3     .SOURCE.COPY1
          3       .SOURCE.COPY2
          3         .SOURCE.COPY3
          6           .SOURCE.IF1
```

Figure 5-1. Preprocessor Listing

5.5.2 Source Listing Generated by SILT2 Phase

The source listing with errors is the section of the listing controlled by the LIST option. For more information on the LIST option, see the *TI Pascal Reference Manual.*

SILT2 attempts to list an error number on the line immediately following the line that contained the error. However, some errors, such as semicolon expected, may not be detected until the first symbol on the next line has been scanned; these errors are indicated one line too late. (Error number 14 in Figure 5-2 indicates that a semicolon is expected after the keyword END on line 12). In other cases, the compiler is not able to identify the actual error; as a result, subsequent code appears to be in error because of an error in the preceding code. Syntax errors are shown in the form !n, where n is the error number. The exclamation point is positioned as closely as possible beneath the symbol in error. Semantic errors are shown in the form **** ERROR # n **** at the beginning of the line following the line on which the error occurs. The message cannot be positioned below the symbol because SILT2 processes an intermediate representation rather than the source code. When errors are detected, SILT2 writes additional error information at the end of the source listing. The information includes the numbers of errors in each category and a table of error numbers and corresponding error messages for each number printed in the listing.

The compilation errors are chained together by lines of the following form:

```
LAST ERROR AT LINE nnnn ON PAGE nnn
```

By starting at the end of the listing and tracing backwards, you can quickly locate every compilation error.

Figure 5-2 and Figure 5-3 present a portion of a source listing that shows error chaining and a line, line 1, inserted by the preprocessor. Figure 5-2 shows a partial source listing with errors. Figure 5-3 is an error-free source listing.

If you specify a print width equal to or greater than 120 characters and set the WIDELIST option to TRUE, the SILT2 listing includes an additional column to the right of the source line as shown in Figure 5-4. This column indicates the nesting level of statements, as well as the duration of routines and FOR, WHILE, REPEAT, CASE, THEN, and ELSE statements.

The nesting level of each statement is indicated by the number following the line number added by the NESTER utility. The routine name appears on the source line with the BEGIN and END statements for the routine. The statement keyword appears on the source line that begins the statement, and the first character of the keyword appears on each subsequent line until the statement is terminated. The keyword pair BEGIN/END marks the duration of statements except that REPEAT/UNTIL and CASE/END delimit REPEAT and CASE statements, respectively.

In Figure 5-4, the body of routine ENTERHASH begins at sequence number 270 and ends on line 500. The keyword WHILE appears on line 320, and the character W appears on each subsequent line down to the END statement on line 430, which terminates the WHILE statement. The characters WTFOR on line 380 indicate the beginning of a FOR statement within a THEN clause within the WHILE statement. In this example, the nesting level is affected by the WHILE, THEN and ELSE statements.

```
 DXPSCL       1.8.0  83.357   TI 990 PASCAL COMPILER   01/07/84 11:01:06
 DIGIO                                                          PAGE   1

   1     (*$MAP,NO OPTIMIZE,WIDELIST*)
   2     PROGRAM DIGIO;
   3     TYPE CHBUF = ARRAY(.1..6.) OF CHAR;
   4     VAR BUFF       : CHBUF;
   5     I,NUM      :INTEGER;
   6     PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
   7     BEGIN     (* CCHAR *)
   8   2 NUM:=0;
   9   3 FOR J = 1 TO I DO
***              !51
  10   4 IF BUFF(.J.)>='0' AND BUFF(.J.)<='9'
  11   5 THEN  NUM:= NUM*10+ORD(BUFF(.J.))-ORD('0')
  12   6 END
  13   7
  14     PROCEDURE CINT (NUM:INTEGER);
***                !14
LAST ERROR AT LINE 9 ON PAGE 1


   MAP OF IDENTIFIERS FOR  CCHAR

IDENTIFIER NAME   KIND         SIZE          STACK                  PICTURE
                             (BYTES,BITS) DISPLACEMENT   (PACKED FIELDS ONLY)
                             LEVEL(DISPL)  (BYTE,BIT)

BUFF             PARAMETER   (12,0)       #0028     DIRECT
NUM              PARAMETER   (2,0)        #0034     INDIRECT
I                PARAMETER   (2,0)        #0036     DIRECT


  15     VAR I          :INTEGER;
  16     BEGIN     (* CINT *)
  17   2 I:= NUM DIV 10;
  18   3 IF I <> 0 THEN CINT(I);
  19   4 WRITE (CHR(NUM MOD 10 + ORD('0')))
  20     END;
          .
          .
          .

NUMBER OF ERRORS = 2
LAST ERROR AT LINE 14 ON PAGE 1

  14   E  ';' EXPECTED
  51   E  ':=' EXPECTED
```

Figure 5-2. Source Listing With Errors

```
DXPSCL    1.8.0    83/357    TI 990 PASCAL COMPILER     01/07/84       11:03:14
DIGIO                                                                    PAGE 1
(*$990,GLOBALOPT,MAP*)                                          PREPROC  83.357
   2    (*$990,MAP,NO OPTIMIZE,WIDELIST*)
   3    PROGRAM DIGIO;
   4    TYPE CHBUF = ARRAY(.1..6.) OF CHAR;
   5    VAR BUFF       : CHBUF;
   6    I,NUM       :INTEGER;
   7    PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
   8    BEGIN     (* CCHAR *)
   9  2 NUM: =0;
  10  3 FOR J : = 1 TO I DO
  11  4 IF BUFF(.J.)>='0' AND BUFF(.J.) <='9'
  12  5 THEN NUM: = NUM*10+ORD(BUFF(.J.))-ORD('0')
  13    END;

   MAP OF IDENTIFIERS FOR CCHAR


IDENTIFIER NAME      KIND          SIZE          STACK            PICTURE
                                (BYTES,BITS)  DISPLACEMENT  (PACKED FIELDS ONLY)
                               LEVEL(DISPL)   (BYTE,BIT)

BUFF              PARAMETER      (12,0)     #0028   DIRECT
NUM               PARAMETER      (2,0)      #0034 INDIRECT
I                 PARAMETER      (2,0)      #0036   DIRECT

  14
  15    PROCEDURE CINT (NUM:INTEGER);
  16    VAR I         :INTEGER;
  17    BEGIN     (* CINT *)
  18  2 I: = NUM DIV 10;
  19  3 IF I <> 0 THEN CINT(I);
  20  4 WRITE (CHR(NUM MOD 10 + ORD('0')))
  21    END;

   MAP OF IDENTIFIERS FOR CINT


IDENTIFIER NAME      KIND          SIZE          STACK            PICTURE
                                (BYTES,BITS)  DISPLACEMENT  (PACKED FIELDS ONLY)
                               LEVEL(DISPL)   (BYTE,BIT)

NUM               PARAMETER      (2,0)      #0028   DIRECT
I                  VARIABLE      (2,0)      #002A   DIRECT
```

Figure 5-3. Source Listing With No Errors (Sheet 1 of 2)

```
 22
 23      BEGIN (* DIGIO *)
 24   2  WRITELN('ENTER 1 TO 5 DIGITS');
 25   3  RESET(INPUT);
 26   4  I: = I+1;
 27   5  WHILE NOT EOLN DO BEGIN  (* INPUT CHARS *)
 28   6  READ (BUFF(.I.));
 29   7  I: = I+1;
 30   8  END;     (* INPUT CHARS *)
 31   9  I : = I-1;
 32   10 CCHAR(BUFF,NUM,I);
 33   11 NUM: = NUM+25;
 34   12 CINT(NUM);
 35   13 WRITELN;
 36      END.

  MAP OF IDENTIFIERS FOR DIGIO

IDENTIFIER NAME     KIND          SIZE              STACK              PICTURE
                                (BYTES,BITS)      DISPLACEMENT      (PACKED FIELDS ONLY)
                                LEVEL(DISPL)       (BYTE,BIT)

BUFF              VARIABLE       (12,0)       #0080      DIRECT
I                 VARIABLE       (2,0)        #008C      DIRECT
NUM               VARIABLE       (2,0)        #008E      DIRECT

37

MAXIMUM NUMBER OF IDENTIFIERS USED = 14

DXPSCL  1.8.0  83.357  OPTIMIZATION SUMMARY

 "CCHAR    " -- 3800 HEAP BYTES REQUIRED TO OPTIMIZE AT LEVEL 0

 "CINT     " -- 3800 HEAP BYTES REQUIRED TO OPTIMIZE AT LEVEL 0

 "DIGIO    " -- 3800 HEAP BYTES REQUIRED TO OPTIMIZE AT LEVEL 0

 INSTRUCTIONS =    33
 CCHAR    LITERALS = 18  CODE =    100  DATA =     58

 INSTRUCTIONS =    21
 CINT     LITERALS = 14  CODE =     78  DATA =     44

 INSTRUCTIONS =    60
 DIGIO    LITERALS = 56  CODE =    238  DATA =    144
```

Figure 5-3. Source Listing With No Errors (Sheet 2 of 2)

```
PREPROC      1.8.0  83.357  PASCAL PREPROCESSOR   01/07/84 11:46:27

SOURCE = PAE8B3.GRAY.TEMP.ENTERH
OBJECT = DS01.OBJECT11
LISTING = PAE8B3.GRAY.LIST.Y
MESSAGE = PAE8B3.GRAY.LIST.SYSMSG
MEM1 = 6,4
MEM2 = 13,4
MEM3 =
PRINT WIDTH = 132
NUMBER OF LINES/PAGE = 60
OPTIONS = (*$990,WIDELIST*)
SUPPRESS PREPROCESSOR LINES = YES

DXPSCL       1.8.0  83.357   TI 990 PASCAL COMPILER   01/07/84 11:46:52
SILT1                                                  PAGE   1

  1   |(*$990,WIDELIST*)                                "PREPROC 83.357  |
  2   |(**                                                                |
  3   |  +         ENTERHAS                                               |
  4   | —       *)                                                        |
  5   |"                                                                  |00000010
  6   |"(C) COPYRIGHT, TEXAS INSTRUMENTS INCORPORATED, 1977.  ALL         |00000020
  7   |"RIGHTS RESERVED.  PROPERTY OF TEXAS INSTRUMENTS INCORPOR-         |00000030
  8   |"ATED.  RESTRICTED RIGHTS - USE, DUPLICATION OR DISCLOSURE         |00000040
  9   |"SUBJECT TO RESTRICTIONS SET FORTH IN TI'S PROGRAM LICENSE         |00000050
 10   |"AGREEMENT AND ASSOCIATED DOCUMENTATION.                           |00000060
 11   |"                                                                  |00000070
 12   |   (*$ NO MAP, WIDELIST *)   (*$ NO TRACEBACK, NO ASSERTS *)       |
 13   |"?COPY COPYFILE                                                    |
 14   |PROGRAM SILT1;                                                     |00000080
 15   |      (*$ NO LIST *)                                               |         SILT1

DXPSCL       1.8.0  83.357   TI 990 PASCAL COMPILER   01/07/84 11:46:52
ENTERHASH                                              PAGE   2

 319   |      ($ LIST, PAGE )                                              |
 320   |PROCEDURE ENTERHASH(VAR IDREC:NAMPTR; HASH:HASHTABRNG);            |00000100
 321   |VAR CURSOR:NAMPTR;                                                 |00000230
 322   |    FOUND : BOOLEAN;                                               |00000240
 323   |    FRONT:NAMPTR;                                                  |00000250
 324   |    I : STRRNG;                                                    |00000260
 325   |BEGIN                                                              |00000270 BEGIN ENTERHAS
 326   |        (FIRST CHECK IF ID IS IN HASH TABLE AT THIS LEVEL ALREADY)  |00000280
 327  3|  CURSOR := HASHTAB[HASH]; FRONT := CURSOR;                         |00000290 0
 328  4|  FOUND := FALSE;                                                  |00000300 0
 329  5| SEARCH :                                                          |00000310 0
 330  6|  WHILE CURSOR <> NIL DO BEGIN                                     |00000320 1 WHILE
 331  7|    IF CURSOR@.NAME < FIRSTNAM THEN ESCAPE SEARCH;                  |00000330 1 W
 332  8| CHECK :                                                           |00000340 1 W
 333  9|    WITH A = CURSOR@, B = IDREC@ DO                                |00000350 1 W
 334 10|      IF A.IDSTRING@[0] = B.IDSTRING@[0] THEN BEGIN                |00000360 2 WTHEN
 335 11|                                    (STRING LENGTHS ARE EQUAL)     |00000370 2 WT
 336 12|        FOR I := 1 TO ORD(B.IDSTRING@[0]) DO                       |00000380 2 WTFOR
 337 13|          IF A.IDSTRING@[I] <> B.IDSTRING@[I] THEN ESCAPE CHECK;   |00000390 2 WT
 338 14|        FOUND := TRUE; END;                                        |00000400 1 W
 339 15|    IF FOUND THEN ESCAPE SEARCH;                                   |00000410 1 W
 340 16|    CURSOR := CURSOR@.LNK;                                         |00000420 1 W
 341 17|    END;                                                           |00000430 0
 342 18|  IF FOUND THEN BEGIN                  (IDENTIFIER DECLARED TWICE) |00000440 1 THEN
 343 19|    ERROR(101); IDREC := CURSOR;                                   |00000450 1 T
 344 20|    END                                                            |00000460 0 T
 345 21|  ELSE BEGIN                 (LINK NEW IDREC INTO HASH TABLE CHAIN)|00000470 1 ELSE
 346 22|    IDREC@.LNK := FRONT; HASHTAB[HASH] := IDREC;                   |00000480 1 E
 347 23|    END;                                                           |00000490 0
 348   |END;                                                               |00000500 END ENTERHAS
 349   |(-                                                              ) |
 350   |BEGIN ($NULLBODY )                                                 |         BEGIN SILT1
 351   |END.                                                               |         END SILT1

MAXIMUM NUMBER OF IDENTIFIERS USED = 215


DXPSCL  1.8.0  83.357   OPTIMIZATION SUMMARY

  "ENTERHAS" -- 6000 HEAP BYTES REQUIRED TO OPTIMIZE AT LEVEL 1
```

Figure 5-4. Source Listing Using WIDELIST and PRINT WIDTH ≥120

5.5.3 Optimization Summary

An optimization summary occurs for each program, function, or procedure that occurs in the source. Each optimization summary consists of two parts. The first part is a collection of error or informative comments. Comments beginning with >>>> are informative. Comments beginning with //// indicate a compiler error, and **** indicates a user error detected by the optimizer.

The second part of an optimization summary is a few lines indicating the number of bytes of heap required to optimize a module.

5.5.4 CODEGEN Summary

The fourth part of a TIP listing is the section that CODEGEN generates. This part also has a summary for each program, function, or procedure in the source. Each summary consists of two parts. The first part is the object listing controlled by the compiler option LISTOBJ. The second part lists the number of bytes of literals, instructions, and stack space the module requires.

5.5.4.1 Object Listing. The object listing is an assembly language listing of the object code produced by the compiler. (Section 11 describes the structure of assembly modules generated by TIP.) Since LISTOBJ is a statement level option, it may be turned on and off inside a module. If LISTOBJ is specified anywhere within a module, the entire literals section is listed as well as the code for the statements affected.

Line numbers appear as a separate comment line filling columns 12 through 45 in the following format:

```
*- - - - - - - - - - - - - - - - - - LINE nn - - - - -
```

Note that these line numbers correspond to the statement line number within the body of a routine. The compiler option WIDELIST enables the inclusion of line numbers in the compiler listing. The second column of numbers contains the statement line numbers.

Because of compiler optimizations, a one-to-one correspondence may not occur between source line numbers and those appearing in the object listing. Also, code for some source lines may appear out of sequence.

5.5.4.2 Example CODEGEN Summary. Figure 5-5 shows an object listing for a procedure called SAMPLE. The assembly language opcode field begins in column 16; the label field starts in column 12. The label LIT$ is always listed as the first label in the module; it is used to reference constants that have been placed in the literals area. The first column of hexadecimal digits in the listing specifies an offset from the beginning of the module. The second column of 4 digits is the hex representation of the instruction or data. Note that externally defined symbols do not have a data value since this value is not known at compile time.

```
DXPSCL      1.8.0  83.357  TI 990 PASCAL COMPILER  01/15/84  11:01:06
SAMPLE                                                        PAGE 2

            IDT     'SAMPLE'            01/15/83  11:01:06
*
*             DXPSCL 1.8.0 83.357
*
          PSEG
      LIT$
0000   53     TEXT 'SAMPLE'              TRACEBACK TEXT STRING
0008 0002     DATA 2                  STATIC NESTING LEVEL
000A 0001     DATA 1
000C 002E'    DATA L002E                EPILOGUE DISPLACEMENT
000E 0000     DATA LIT$                LITERALS AREA
*
              DEF SAMPLE
              REF ENT$S
              REF RET$S
*
          SAMPLE
0010 06A0     BL   @ENT$S
     ----
0014 002E     DATA >002E
          *------------------ LINE 2 -----
0016 C1A9     MOV  @40(R9),R6
     0028
001A 0226     AI   R6,7
     0007
001E CA46     MOV  R6,@42(R9)
     002A
          *------------------ LINE 3 -----
0022 C1E9     MOV  @44(R9),R7
     002C
0026 1603     JNE  L002E
          *------------------ LINE 4 -----
0028 CA60     MOV  @LIT$+>000A,@42(R9)
     000A'
     002A
          L002E
          *------------------ LINE 4 -----
002E 0460     B    @RET$S
     ----
0032

  INSTRUCTIONS = 8
  SAMPLE LITERALS = 16 CODE = 34 DATA = 46
```

Figure 5-5. Sample Object Listing

5.5.5 Cross-Reference

The final part of the listing is the optional cross-reference. The cross-reference listing is for the entire source module. Each symbol is listed in alphabetical order, but only the first 10 characters of a symbol are listed. The line numbers at which the symbol can be found appear to the right of the symbol. These numbers match those generated by the WIDELIST option in the leftmost column of the source listing. The plus sign (+) after a line number indicates that the symbol appears more than once on that line. A sample cross-reference listing is as follows:

```
                    PASCAL CROSS-REFERENCE UTILITY

I                   4   6   7
J                   3   7   8+
SAMPLE              2

3  IDENTIFIERS                      8  OCCURRENCES
```

5.6 MESSAGE FILE DESCRIPTION

The message file indicates the amount of stack and heap used for each of the compiler tasks. The message file for a TIP compilation has six sections. The first section shows the execution of the preprocessor and contains any error messages that the preprocessor generates. The next four subsections list the name of the program, procedure, or function after the module is processed. The second section shows the execution of SILT1. The third section shows the execution of SILT2. If compilation errors are detected, an error message will appear before the name of the module that contains the errors. The fourth section shows the execution of the optimizer. The fifth section shows the execution of the code generator. The final section shows the execution of the cross-reference task. Figure 5-6 provides a sample message file.

```
PREPROCE EXECUTION BEGINS
NORMAL TERMINATION
STACK USED = 4548 HEAP USED = 612

SILT1 EXECUTION BEGINS
SAMPLE
NORMAL TERMINATION
STACK USED = 4054 HEAP USED = 1968

SILT2 EXECUTION BEGINS
NONFATAL ERRORS IN RUNTIME
SAMPLE
NONFATAL ERRORS IN PROGRAM
NORMAL TERMINATION
STACK USED = 10196 HEAP USED = 1666

T9OPT EXECUTION BEGINS
SAMPLE
NORMAL TERMINATION
STACK USED = 3648 HEAP USED = 2472

CODEGEN EXECUTION BEGINS
SAMPLE
NORMAL TERMINATION
STACK USED = 8592 HEAP USED = 892

PSCLXREF EXECUTION BEGINS
NORMAL TERMINATION
STACK USED = 14272 HEAP USED = 294
```

Figure 5-6. Message File

5.7 COMPILER MEMORY USAGE

SILT1 and SILT2 stack sizes depend on the maximum nesting level of syntactic constructs. (It is a recursive descent parser.) This includes the nesting of statements and routines, but is not affected by the complexity of expressions since they are parsed in a special way. If SILT needs more than the default amount of stack, it is usually because the program has an unusual depth of routine nesting. (Note: the stacks include a number of fixed-sized tables, so the size is not proportional to the nesting.)

The SILT1 heap is mostly filled with identifier information. Each identifier uses space equal to the number of characters in the name, plus 11 or 12 bytes of attribute information and overhead. The maximum amount of heap used depends on the maximum number of identifiers whose extent includes any particular point in the program.

The SILT2 heap contains mostly type information. Each non-identical type uses from 6 to 18 bytes. For example, the Pascal declaration:

```
VAR X:  ARRAY[1..100] OF INTEGER;
```

creates two new unique types: the subrange 1..100 and the array type. Thus, SILT2 heap space can be conserved by using named types for any types which appear more than once. Here also, the space used depends on the number of types with overlapping extents.

The optimizer uses a fixed amount of stack space. The optimizer uses the heap to hold all of the code for a single routine, so the heap space is directly proportional to the size of the largest routine body.

The CODEGEN stack is mostly fixed-size tables. What little variation in stack usage there is depends mostly on the complexity of expressions since recursive code is used to traverse expression trees. Note that use of the check options adds a lot of hidden complexity.

The CODEGEN heap is used to hold a linked list representing the label and jump structure of a single routine. Thus the heap usage is roughly proportional to the size of the largest routine body. Note that T9OPT and CODEGEN only work on one routine at a time, so things like routine nesting and amount of global declarations are not relevant.

SILT allows a maximum number of 1023 identifiers active at one time. This number includes 55 pre-defined identifiers but does not include constant and enumeration value names. The message "MAXIMUM NUMBER OF IDENTIFIERS USED = _____" on the listing can be checked to see if the limit is being approached. The number shown on the listing does not include the pre-defined identifiers, so it is limited to 968.

6

Separate Compilation

6.1 GENERAL

Development of a large program is significantly less expensive when modules of the program can be changed and recompiled without recompiling the entire program. In a block-structured language such as TIP, separate compilation is more difficult than in assembly language or high-level languages that are not structured. The difficulty results from the scope rules of TIP and the capability of passing parameters either by value or by reference. To separately compile a TIP routine, all global declarations must be included in the source code so that the environment is identical to that in which the routine executes. In this context, global declarations include the declaration sections of all routines within which the routine is nested. The process of manually merging the declaration sections is tedious and error-prone.

This section describes the TIP software that separately compiles TIP program modules. Consult the *TI Pascal Configuration Processor Tutorial* for step-by-step instructions on how to use the configuration processor. This section and the tutorial contain complementary information and should be used together.

6.2 REQUIREMENTS FOR SEPARATE COMPILATION

The TIP compiler produces a separate object module for each program and for each routine of the program. Two object modules result from compiling the following code:

```
PROGRAM A;
VAR X,Y,Z:INTEGER;
PROCEDURE B(W:INTEGER);FORWARD;
    PROCEDURE B;
    BEGIN (*B*)
      W:=Y
    END (*B*)
BEGIN (*A*)
  B(X)
END (*A*).
```

The compiler concatenates the two object modules in a single file. Alternatively, the two modules can be separated (using the Text Editor, for example) and stored as separate library members. Also, they can be concatenated before link editing, or can be specified to the Link Editor by using an INCLUDE command in the control file.

When the main program module A needs to be recompiled, a new module for A can be compiled and the new module linked with the existing module for B. To recompile a new module for A, simply omit the code for procedure B and compile the following:

```
PROGRAM A;
VAR X,Y,Z:INTEGER;
PROCEDURE B(W:INTEGER);FORWARD;
BEGIN (*A*)
      B(X)
END (*A*).
```

Since procedure B is omitted, the forward declaration of procedure B must be included so that the call to procedure B in program A results in the correct linkage.

To correctly recompile routine B, the compiler must have the declarations of the main program as well as those of routine B. The source code is as follows:

```
PROGRAM A;
VAR X,Y,Z:INTEGER;
PROCEDURE B(W:INTEGER);FORWARD;
    PROCEDURE B;
    BEGIN (*A*)
      W:=Y
    END (*B*);
BEGIN (*$NO OBJECT*)
END (*A*).
```

The NO OBJECT option suppresses the production of an object module for A, and only the object module for B is produced. This module can be linked with the existing module for A to obtain a new version of the entire program.

The NO OBJECT option is required since, without it, the presence of the BEGIN and END keywords for module A would produce an object module. If the object module for A were created, it would be necessary to delete it in order to properly link the existing module A with the new module B.

A different approach to the problem of separately compiling a program module is to store individual source modules in a library. The source code in the example can be separated as follows:

```
PROGRAM A;
VAR X,Y,Z:INTEGER;
PROCEDURE B(W:INTEGER);FORWARD;
BEGIN (*A*)
  B(X)
END(*A*).

PROCEDURE B;
BEGIN (*B*)
  W:=Y
END (*B*);
```

The first of the two source modules shown can be used without alteration to recompile module A. The two must be combined by appropriate text editing to recompile module B.

The process of separately compiling a program with more than two routines or with routines nested to two or more levels is rather complex. The declarations of the main program must be included, along with the declarations of all routines in which the routine to be separately compiled is nested. On the other hand, only the statement section of the routine being separately compiled is included. Although source code can be manually prepared for separate compilation of a routine, TIP software includes the configuration processor (CONFIG) to perform these operations.

6.3 THE CONFIGURATION PROCESSOR

The configuration processor supports separate compilation of TIP modules by performing the following functions:

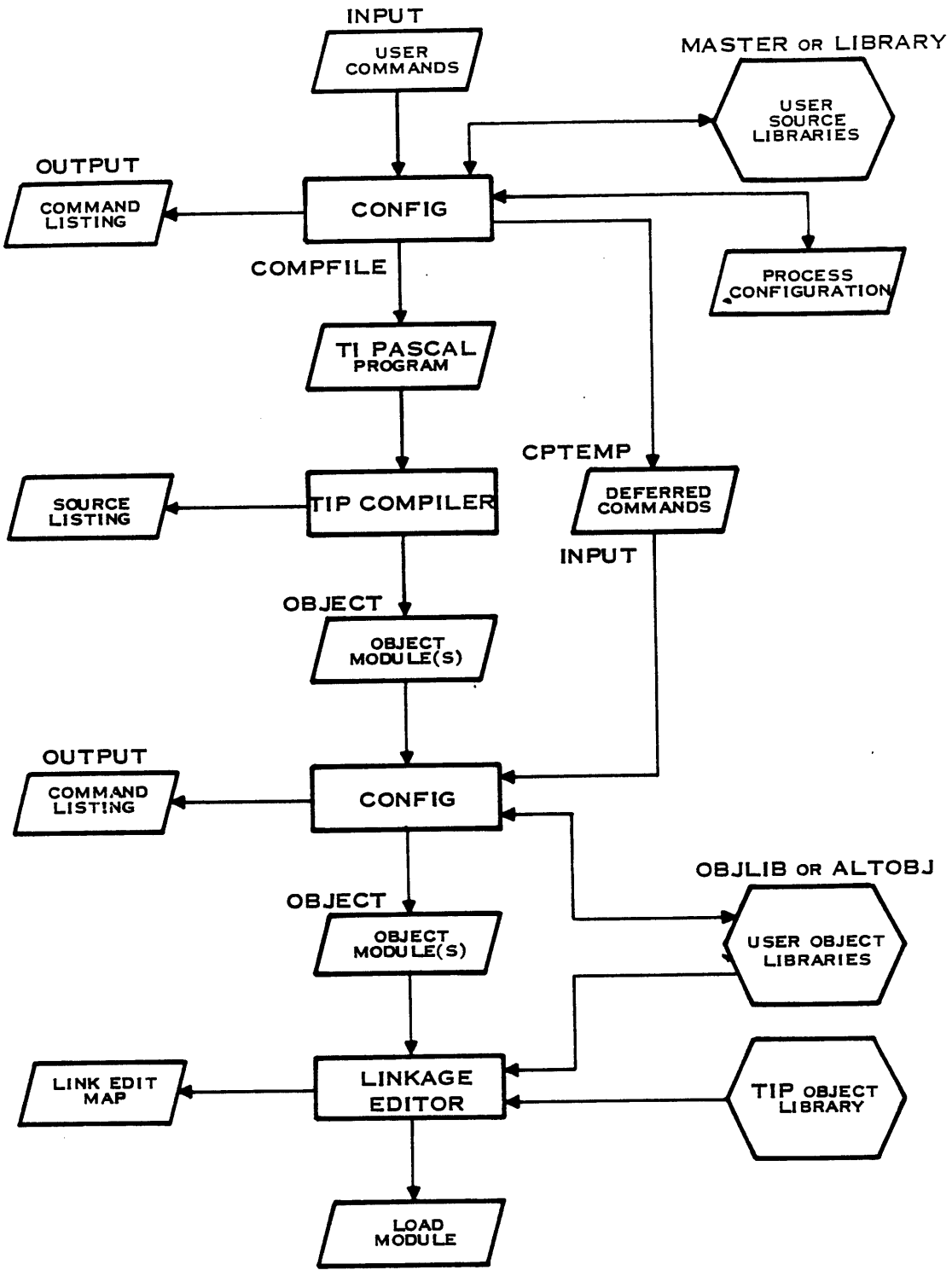
- Maintaining a library of program source modules to be combined as needed to separately compile each module of a program
- Preparing a source program for each separate compilation
- Maintaining a library of object modules of a program, from which appropriate object modules are linked

6.3.1 Functional Description of CONFIG

Figure 6-1, a flowchart of the separate compilation operation, illustrates the functions of CONFIG. The following description assumes that your source libraries include source modules, as required for separate compilation. You can write the source code as separate modules in a source library, or you can create the library from a source program by using the source Split Program utility (SPLITPGM).

To perform separate compilation using CONFIG, first execute CONFIG using the command XCONFIG (for background execution), or XCONFIGI (for interactive execution). In response to your commands, CONFIG communicates with your source libraries and prepares the desired TIP program. CONFIG also writes a process configuration as specified in your commands; this configuration describes the hierarchical structure of the program. Next, CONFIG writes a file of deferred commands for a subsequent execution of CONFIG and a command listing file that contains the commands and a copy of the process configuration.

The TIP compiler processes TIP source written by CONFIG. The compiler produces a separate object module for each routine being compiled, as well as a source listing. CONFIG then executes again, using the commands in the deferred command file (written during the previous run of CONFIG). The object modules written by the compiler are concatenated on a single file; CONFIG separates the modules, writing them as members of your object libraries. Optionally, CONFIG can collect a full set of object modules to be supplied to the Link Editor. The Link Editor links the object modules with modules from the TIP object library to form a load module (linked object module) and writes the link edit map listing.



2277728

Figure 6-1. Flow of Separate Compilation Using CONFIG

6.3.2 Format of Source Modules

Source modules for input to CONFIG must be separated and stored as members of your source libraries. They must conform to the following rules:

- A source module consists of one program, procedure, or function in which all contained procedures and functions have been replaced by forward declarations.
- Each procedure and function must be declared in a forward declaration to ensure that each calling sequence is correctly defined.
- Keyword BEGIN of the compound statement that contains the statements of the program, procedure, or function is in character positions 1 through 5. The component statements must be indented.
- Keyword END of the compound statement that contains the statements of the program, procedure, or function is in character positions 1 through 3. The component statements must be indented.
- Compiler option NULLBODY must not be specified in any of the source modules.
- Character position 1 must not contain an asterisk (*).
- Character position 1 must not contain a minus sign (–) unless character position 2 also contains a minus sign.
- Within the declaration section, a comment that begins in character position 1 must be closed by a brace (}) in character position 72 or by an asterisk and parenthesis (*)) in character positions 71 and 72 of the same or a succeeding line.

CONFIG recognizes one or more comments in the declaration section of a module preceding the TYPE or VAR declaration as the documentation section of the module. Comments in this section must begin in character position 1 and close in character position 72 of the same or a succeeding line; also these comments can be listed separately from the source code.

The Nester utility can be used to comply with indentation requirements (third and fourth rules). SPLITPGM can be used to divide a source program into source modules in accordance with the first rule.

6.3.3 Configuration Processor Commands

The operation of the configuration processor is controlled by commands entered on file INPUT. Each command begins with an asterisk (*). The commands are free format, meaning that they can appear in any column, can extend over more than one line, or several commands can appear on a single line. The command file is terminated by an end-of-file (EOF). When entering commands interactively from a VDT, EOF is signaled by pressing the Enter key.

In some BNF productions for configuration processor commands, angle brackets (< >) are used as terminal symbols. When an angle bracket in a BNF production is a terminal symbol, it is enclosed in quotation marks (" ").

Many of the commands include a <location>, which is defined as follows:

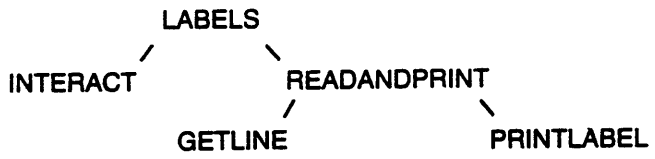
<location> ::= "<" [<library>,] <member> ">"

The location consists of a library synonym and a member name.

6.3.4 Process Configuration

CONFIG determines the structure of the program. The structure includes the name of the main program, the names of the routines, and the name of the routine within which each routine is declared (or the main program name for global routines). The primary data structure that contains this information is called the *process configuration.* CONFIG writes, maintains, and uses this structure. Via user commands, you specify the structure and contents of the process configuration.

The process configuration is structured like a tree, with each node representing a source module of the program. The root node represents the main program module. The following is an example:



The process configuration is represented in tabular form, as follows:

PROCESS NAME	SOURCE LOCATION	OBJECT LOCATION	FLAGS SET
LABELS	<LIBRARY ,LABELS>		
INTERACT	<LIBRARY ,INTERACT>		
READANDP	<LIBRARY ,READANDP>		
GETLINE	<LIBRARY ,GETLINE>		
PRINTLAB	<LIBRARY ,PRINTLAB>		

All of the routines are descendents of the main program. Routines INTERACT and READANDP are sons of LABELS; routines GETLINE and PRINTLAB are sons of READANDP. If another routine were nested in PRINTLAB, it would be the son of PRINTLAB and a descendent of READANDP.

The example process configuration corresponds to the commands in the example in paragraph 6.3.4.4. The source locations listed are the locations at which CONFIG accesses the source modules. The library name is the default value (paragraph 6.3.9) because no DEFAULT SOURCE command (paragraph 6.3.9.6) has been entered. No object locations are listed because no DEFAULT OBJECT command (paragraph 6.3.9.7) has been entered. No flags (paragraph 6.3.7) are listed because the initial states of the flags have not been altered.

The following commands define process configurations:

- *BUILD PROCESS
- *ADD
- *CAT PROCESS

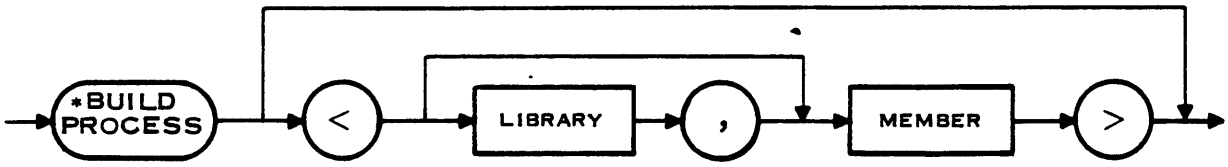
6.3.4.1 BUILD PROCESS Command. The BUILD PROCESS command initializes a configuration process as the current configuration process. The syntax of the command is as follows:

<build process command> ::= *BUILD PROCESS[<location>]

<location> ::= "<" [<library>,] <member> ">"

The syntax diagram is as follows:

BUILD PROCESS Command:



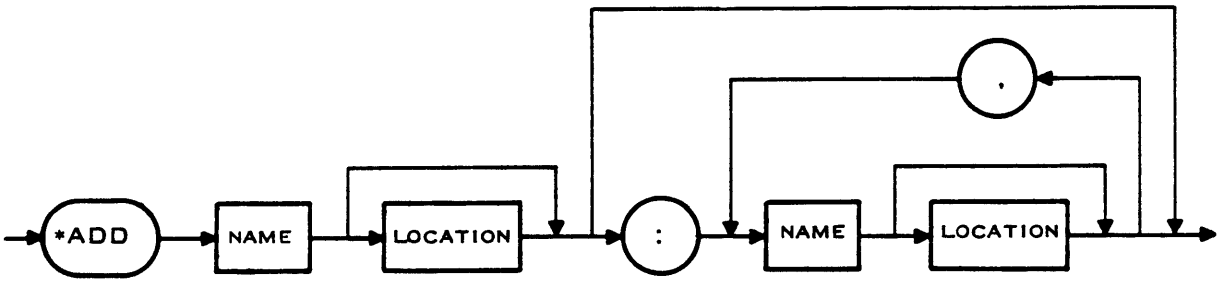
The location parameter is optional. It may be omitted when the location is specified in the CAT PROCESS command. The location consists of a library synonym and a member name. The library synonym may be omitted in which case the default source library (LIBRARY, unless it has been altered by a DEFAULT SOURCE command) is used.

6.3.4.2 ADD Command. The ADD command specifies the name of the root node for a process configuration and, optionally, a location at which the source module for the node is cataloged. An alternate form of the command specifies the name and location of one or more nodes as sons of a specified node. The syntax of the command is as follows:

<add command> ::= *ADD<name> [<location>][:<name>[<location>]
{,<name>[<location>]}]

The syntax diagram is as follows:

ADD Command:



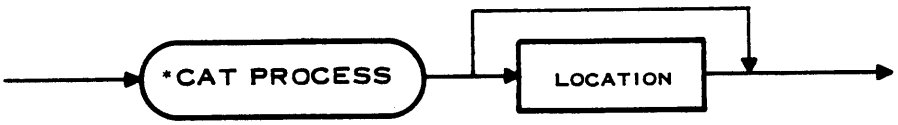
The first ADD command following a BUILD PROCESS command has only one name, that of the root node of a process configuration. The location parameter is the location as defined for the BUILD PROCESS command and is optional. When no location is entered, the node name is used as the member of the default source library. Subsequent ADD commands require that the first name parameter be the name of a previously defined node. The location, if entered, is ignored. Name parameters to the right of the colon (:) define additional nodes that are sons of the node named in the first name parameter. The location, if entered, specifies a library and member or a member of the default source library for the module. If no location is entered, the node name is used as the member name of the default source library (LIBRARY, unless it has been altered by a DEFAULT SOURCE command).

6.3.4.3 CAT PROCESS Command. The CAT PROCESS command causes the current process to be stored at the specified location. The syntax of the command is as follows:

<cat process command> ::= *CAT PROCESS [<location>]

The syntax diagram is as follows:

CAT PROCESS Command:



The location is the location as defined for the BUILD PROCESS command. When a location has been previously specified for the current process configuration (in a BUILD PROCESS or USE PROCESS command), the location may be omitted. When a location is specified, the location in the CAT PROCESS command applies, replacing any previous location.

6.3.4.4 Process Configuration Command Example. The following is an example of a set of commands that define the process configuration described in paragraph 6.3.3:

```
*BUILD PROCESS
*ADD LABELS
*ADD LABELS:   INTERACT
*ADD LABELS:   READANDP
*ADD READANDP:   GETLINE
*ADD READANDP:   PRINTLAB
*CAT PROCESS<LIBRARY,PROCESS>
```

The BUILD PROCESS command initializes a process configuration, and the first ADD command defines the root node of the structure as the main program, LABELS. The next two ADD commands define INTERACT and READANDP as sons of the root node LABELS. The last two ADD commands define GETLINE and PRINTLAB as sons of READANDP. The CAT PROCESS command specifies that the process is cataloged as member PROCESS of a library with a pathname that is the value of synonym LIBRARY. Since the BUILD PROCESS command and did not specify a location for the process configuration, the CAT PROCESS command requires a location parameter.

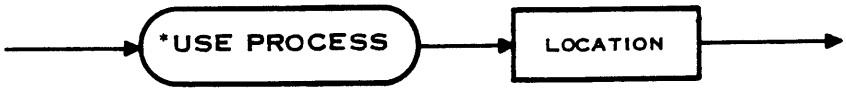
Figure 6-3 shows the source code for the program structure used in the process configuration example.

6.3.4.5 USE PROCESS Command. The USE PROCESS command specifies an existing process configuration as the current process configuration. The syntax of the command is as follows:

<use process command> ::= *USE PROCESS <location>

The syntax diagram is as follows:

USE PROCESS Command:



The location is the location as defined for the BUILD PROCESS command. The location must be specified and must be the location of a cataloged process configuration. The process configuration becomes the current process configuration for the remainder of the run. Either a USE PROCESS command or a BUILD PROCESS command must define a current process configuration before any command that operates on a process configuration is entered.

6.3.5 Compilation

The principal use of CONFIG is to compile a program or to separately compile one or more object modules of a program. As Figure 6-1 shows, compilation begins with an execution of CONFIG that produces a source module and a file of deferred commands for a subsequent pass of CONFIG. The source module includes the necessary source library members from which the required module(s) may be compiled. The compiler executes, using the source module written by CONFIG, and provides an object file containing the desired module(s). A second execution of CONFIG, using the deferred command file, separates the object file into an object module library and/or writes an object file for direct input to the Link Editor.

The user commands for compilation may define a process configuration or specify a previous configuration. A BUILD PROCESS command, ADD commands, and a CAT PROCESS command define a process configuration. A USE PROCESS command specifies a previously built process configuration. A COMPILE command is required to specify the source modules to be compiled.

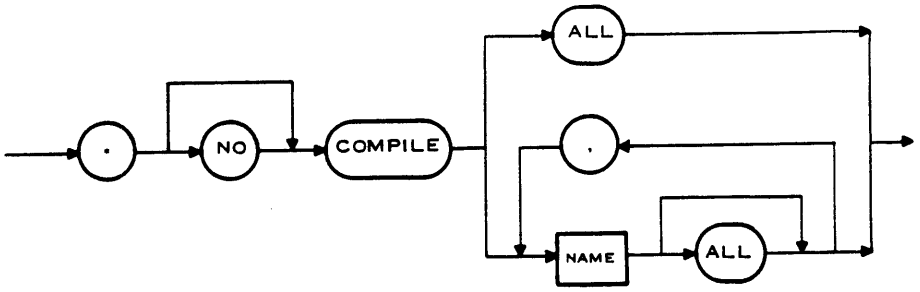
The deferred command file includes a USE PROCESS# command, a SPLIT OBJECT command, and an EXIT command. The USE PROCESS# command in the beginning of the deferred command file is different from the USE PROCESS user command. In the deferred command file, the USE PROCESS# command is followed by an external representation of the process configuration, placed there by CONFIG. This mechanism is the one by which CONFIG passes the entire process configuration to the succeeding run of CONFIG. Consequently, the second run of CONFIG produces object modules that correspond to the source modules from the first CONFIG run.

6.3.5.1 COMPILE Command. The COMPILE command causes CONFIG to prepare a source module for compilation and specifies the module(s) to be compiled. The syntax for the command is as follows:

<compile command> ::= *[NO] COMPILE ALL
| *[NO] COMPILE<name>[ALL] {,<name>[ALL]}

The syntax diagram is as follows:

COMPILE Command:



The optional keyword NO allows you to inhibit compilation of the module(s) named in the command. When the keyword ALL follows the keyword COMPILE with no name parameter, the entire program is compiled. The name parameter is the name of the module to be compiled. When the keyword ALL follows a name parameter, the named module and all its descendents are compiled. Additional name parameters may be entered, and each may be followed by the keyword ALL.

Use the following guidelines when selecting modules for recompilation:

- When a statement within the compound statement of a program or routine is changed, recompile the module that contains the program or routine.
- When a global declaration of a program is changed, recompile the entire program.
- When a declaration of a routine is changed, recompile the module that contains the declaration and all modules that are its descendents.

6.3.5.2 SPLIT OBJECT Command. The SPLIT OBJECT command is placed in the deferred command file when a COMPILE command is included in the file INPUT. You should not enter the SPLIT OBJECT command. The format of the command is as follows:

```
*SPLIT OBJECT
```

The SPLIT OBJECT command causes CONFIG to catalog each module of the object file written by the compiler as a member of a library. The library synonym and/or member name may be specified in a USE OBJECT command or a DEFAULT OBJECT command. Otherwise, the default object library is used with the node name as the member name. The default object library is ALTOBJ.

The SPLIT command used on individual modules means that if the COMPILE flag is on, then the object file will include object code for the specified module (or modules), and it should be split. NO SPLIT has the same effect as NO COMPILE.

6.3.5.3 EXIT Command. The EXIT command is the last command placed in the deferred command file. The format of the command is as follows:

```
*EXIT
```

When CONFIG reads the EXIT command in the file INPUT, it terminates processing. You can enter the command to abort execution of CONFIG. No files are saved.

6.3.5.4 Compilation Examples. The following commands in the file INPUT cause CONFIG to provide a source file that contains all source modules for a program, as well as a deferred command file to catalog all object modules in the object file written by the compiler:

```
*USE PROCESS <LIBRARY,PROCES>
*COMPILE ALL
```

In this example, the process configuration has been built previously and cataloged at location <LIBRARY,PROCES>. If the file INPUT includes a BUILD PROCESS command, a set of ADD commands, and a CAT PROCESS command instead of the USE PROCESS command, a new process configuration is built and cataloged. The COMPILE command applies to the program that corresponds to the current process configuration, whether the process configuration was built in a previous run of CONFIG or in the same run.

Figure 6-2 shows the contents of file OUTPUT following the initial run of CONFIG. The commands are listed first, followed by a tabular representation of the process configuration. The flags set in the process configuration are set by the COMPILE command. The pathnames corresponding to the files are listed next, followed by the pathnames assigned to the library synonyms defined by CONFIG.

Figure 6-3 shows the source listing of the compiler run. Notice that CONFIG has inserted some comment lines. Otherwise, the program is identical to the Nester output of the same source program.

Figure 6-4 shows the contents of file OUTPUT for the deferred processing run of CONFIG. The deferred commands are listed, as are the object modules. The node name and location are shown for each module, followed by the termination record of the module.

```
DXPSCLCP   1.8.0  83.357    TI 990 CONFIGURATION PROCESSOR  12/28/83          11:16:57
*USE PROCESS <LIBRARY,PROCESS>
*COMPILE ALL

PROCESS NAME      SOURCE LOCATION    OBJECT LOCATION    FLAGS SET
-------------------------------------------------------------------
LABELS            <LIBRARY ,LABELS >                      0 1
  INTERACT        <LIBRARY ,INTERACT>                     0 1
  READANDP        <LIBRARY ,READANDP>                     0 1
    GETLINE       <LIBRARY ,GETLINE >                     0 1
    PRINTLAB      <LIBRARY ,PRINTLAB>                     0 1

INPUT     = ST16
CRTFIL    = ST16
OUTPUT    = .OUTPUT16
COMPFILE  = .COMPFI16
CPTEMP    = .CPTEMP16
OBJECT    = .OBJECT16

MASTER    = .MASTER16
LIBRARY   = SYS2.DP0020.SRCLIB
OBJLIB    = .OBJLIB16
ALTOBJ    = .ALTOBJ16
```

Figure 6-2. Contents of File OUTPUT, Initial CONFIG Run, Full Compilation

```
(**         LABELS
   +           INTERACT
   ,           READANDP
     +           GETLINE
     ,           PRINTLAB
  ---     *)
PROGRAM LABELS;                                                          0000010
(*-------------------------------------------------------------------------0000020
   PROGRAM LABELS:                                                       0000030
   PURPOSE :     THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE  0000040
                 COPIES OF THAT LABEL.                                   0000050
   FILES USED:   INPUT - FOR USER-SUPPLIED PARAMETERS AND THE LABEL      0000060
                 CPTFIL - USED FOR PROMPTING INPUT                       0000070
                 OUTPUT - MULTIPLE COPIES OF THE LABEL                   0000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                            0000090
-------------------------------------------------------------------------*) 0000100
VAR CRTFIL : TEXT ;                              (*USED TO PROMPT INPUT*)  0000110
    CHARSPERLINE : INTEGER;                (*NUMBER OF CHARACTERS PER LINE*)  0000120
    LINESPERLABEL : INTEGER;                   (*NUMBER OF LINES PER LABEL*)  0000130
    COPYCOUNT : INTEGER;                       (*NUMBER OF COPIES TO PRINT*)  0000140
PROCEDURE INTERACT; FORWARD;                                             0000150
PROCEDURE READANDPRINT; FORWARD;                                         0000160
(**                                                                    *)
PROCEDURE INTERACT;                                                      0000010
(*-------------------------------------------------------------------------0000020
  PROCEDURE INTERACT;                                                    0000030
  PURPOSE :  INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.       0000040
  OUTPUTS :  CHARASPERLINE - NUMBER OF CHARACTERS PER LINE               0000050
             LINESPERLABEL - NUMBER OF LINES PER LABEL                   0000060
             COPYCOUNT - NUMBER OF LABELS TO PRINT                       0000070
-------------------------------------------------------------------------*) 0000080
BEGIN                                                    (*INTERACT*)    0000090
  REWRITE( CRTFIL );                                                     0000100
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                    0000110
  RESET(INPUT); READ( CHARSPERLINE );                                    0000120
  WRITELN(CRTFIL, 'HOW MANY LINES PER LABEL?' );                         0000130
  READLN; READ( LINESPERLABEL );                                         0000140
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                 0000150
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL' );    0000160
END;                                                     (*INTERACT*)    0000170
(*,                                                                  *)
PROCEDURE READANDPRINT;                                                  0000010
(*-------------------------------------------------------------------------0000020
  PROCEDURE READANDPRINT;                                                0000030
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.  0000040
  PROCEDURES CALLED : GETLINE, PRINTLABEL                                0000050
-------------------------------------------------------------------------*) 0000060
```

Figure 6-3. Source Listing, Full Compilation Example (Sheet 1 of 2)

```
TYPE                                                                    0000070
    LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                    0000080
VAR                                                                     0000090
    LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                    0000100
PROCEDURE GETLINE (VAR THISLINE : LINE); FORWARD;                       0000110
PROCEDURE PRINTLABEL; FORWARD;                                          0000120
(*+                                                                    *)
PROCEDURE GETLINE (*VAR THISLINE : LINE*);                              0000010
(*--------------------------------------------------------------------- 0000020
  PROCEDURE GETLINE;                                                    0000030
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.                     0000040
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                0000050
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.                          0000060
---------------------------------------------------------------------*) 0000070
VAR CH : INTEGER;                                                       0000080
BEGIN                                                    (*GETLINE*)    0000090
  READLN; CH := 1;                                                      0000100
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN                 0000110
    READ( THISLINE(.CH.) ); CH := CH + 1;                               0000120
    END;                                 (*FILL IN REST OF LINE WITH BLANKS*)  0000130
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := '';                   0000140
END;                                                     (*GETLINE*)    0000150
(*,                                                                    *)
PROCEDURE PRINTLABEL;                                                   0000010
(*--------------------------------------------------------------------- 0000020
  PROCEDURE PRINTLABEL;                                                 0000030
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.                    0000040
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL                   0000050
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                0000060
            LABELIMAGE - THE LABEL TO BE PRINTED                        0000070
---------------------------------------------------------------------*) 0000080
BEGIN                                                 (*PRINTLABEL*)    0000090
  FOR L := 1 TO LINESPERLABEL DO BEGIN                                  0000100
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );      0000110
    WRITELN; END;                                                       0000120
END;                                                  (*PRINTLABEL*)    0000130
(*-                                                                    *)
BEGIN                                              (*READANDPRINT*)     0000130
  FOR L := 1 TO LINSPERLABEL DO GETLINE( LABELIMAGE(.L.) );             0000140
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;                                0000150
END;                                               (*READANDPRINT*)     0000160
(*-                                                                    *)
BEGIN                                                    (*LABELS*)     0000170
  INTERACT; READANDPRINT;                                               0000180
END.                                                     (*LABELS*)     0000190
```

Figure 6-3. Source Listing, Full Compilation Example (Sheet 2 of 2)

```
DXPSCLCP   1.8.0  83.357    TI 990 CONFIGURATION PROCESSOR  12/28/83          12:10:10
*USE PROCESS #
*SPLIT OBJECT
INTERACT = <ALTOBJ  ,INTERACT >:          INTERACT     12/28/83   12:13:02   DXPSCL
GETLINE  = <ALTOBJ  ,GETLINE  >:          GETLINE      12/28/83   12:13:14   DXPSCL
PRINTLAB = <ALTOBJ  ,PRINTLAB >:          PRINTLAB     12/28/83   12:13:24   DXPSCL
READANDP = <ALTOBJ  ,READANDP >:          READANDP     12/28/83   12:13:38   DXPSCL
LABELS   = <ALTOBJ  ,LABELS   >:          LABELS       12/28/83   12:13:52   DXPSCL
*EXIT
```

Figure 6-4. Contents of File OUTPUT, Deferred Processing, Full Compilation

A partial compilation of two modules of the same program is performed by using commands such as the following:

```
*USE PROCESS <LIBRARY,PROCES>
*COMPILE PRINTLAB,INTERACT
```

The COMPILE command for this example specifies compiling modules INTERACT and PRINTLAB. The source module for the compilation requires the declarations for routines LABELS and READANDPRINT, the ancestors of INTERACT and PRINTLABEL. The source module also must include both the declarations and statements of routines INTERACT and PRINTLABEL.

Figure 6-5 shows the contents of file OUTPUT for the initial CONFIG run. As in the full compilation example, the file contains the commands and a tabular representation of the process configuration. The flags set in the process configuration correspond to the portions of modules that are to be combined in the source file that CONFIG builds. The pathnames are identical to those in the preceding example.

Figure 6-6 shows the source listing of the compiler run. The declaration portion of module LABELS is first, including the forward declarations of routines INTERACT and READANDPRINT. Next is module INTERACT, followed by the declaration portion of module READANDPRINT. Forward declarations of routines GETLINE and PRINTLABEL are included, even though no portion of module GETLINE is included in the partial compilation. Module PRINTLABEL is next, followed by the statement portions of routine READANDPRINT and program LABELS. To inhibit the compiler from writing modules READANDP and LABELS, the statement portions of these modules supplied by CONFIG consist of BEGIN keywords followed by NULLBODY option comments and END keywords.

```
DXPSCLCP   1.8.0  83.357  TI 990 CONFIGURATION PROCESSOR  12/28/83          13:28:37
*USE PROCESS <LIBRARY, PROCESS>
*COMPILE PRINTLAB, INTERACT

PROCESS NAME            SOURCE LOCATION            OBJECT LOCATION          FLAGS SET
------------            ---------------            ---------------          ---------
LABELS                 <LIBRARY ,LABELS  >                                  0
  INTERACT             <LIBRARY ,INTERACT>                                  0 1
  READANDP             <LIBRARY ,READANDP>                                  0
    GETLINE            <LIBRARY ,GETLINE >
    PRINTLAB           <LIBRARY ,PRINTLAB>                                  0 1


INPUT    = ST16
CRTFIL   = ST16
OUTPUT   = SYS2.DP0020.LIST
COMPFILE = .COMPFI16
CPTEMP   = .CPTEMP16
OBJECT   = .OBJECT16

MASTER   = .MASTER16
LIBRARY  = SYS2.DP0020.SRCLIB
OBJLIB   = SYS2.DP0020.SRCLIB
ALTOBJ   = SYS2.DP0020.SRCLIB
```

Figure 6-5. Contents of File OUTPUT, Initial Run, Partial Compilation

```
(*+
   +            INTERACT
   /
    +             PRINTLAB
---       *)
PROGRAM LABELS;                                                              0000010
(*-------------------------------------------------------------------------- 0000020
   PROGRAM LABELS:                                                           0000030
   PURPOSE :     THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE      0000040
                 COPIES OF THAT LABEL.                                       0000050
   FILES USED:   INPUT - FOR USER-SUPPLIED PARAMETERS AND THE LABEL          0000060
                 CPTFIL - USED FOR PROMPTING INPUT                           0000070
                 OUTPUT - MULTIPLE COPIES OF THE LABEL                       0000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                                0000090
--------------------------------------------------------------------------*) 0000100
VAR CRTFIL : TEXT ;                                (*USED TO PROMPT INPUT*)  0000110
    CHARSPERLINE : INTEGER;                (*NUMBER OF CHARACTERS PER LINE*)  0000120
    LINESPERLABEL : INTEGER;                   (*NUMBER OF LINES PER LABEL*)  0000130
    COPYCOUNT : INTEGER;                       (*NUMBER OF COPIES TO PRINT*)  0000140
PROCEDURE INTERACT; FORWARD;                                                 0000150
PROCEDURE READANDPRINT; FORWARD;                                             0000160
(*+                                                                       *)
PROCEDURE INTERACT;                                                          0000010
(*-------------------------------------------------------------------------- 0000020
  PROCEDURE INTERACT;                                                        0000030
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.            0000040
  OUTPUTS : CHARASPERLINE - NUMBER OF CHARACTERS PER LINE                    0000050
            LINESPERLABEL - NUMBER OF LINES PER LABEL                        0000060
            COPYCOUNT - NUMBER OF LABELS TO PRINT                            0000070
--------------------------------------------------------------------------*) 0000080
BEGIN                                                    (*INTERACT*)        0000090
  REWRITE( CRTFIL );                                                         0000100
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                        0000110
  RESET(INPUT); READ( CHARSPERLINE );                                        0000120
  WRITELN(CRTFIL, 'HOW MANY LINES PER LABEL?' );                             0000130
  READLN; READ( LINESPERLABEL );                                             0000140
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                     0000150
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');         0000160
END;                                                     (*INTERACT*)        0000170
```

Figure 6-6. Source Listing, Partial Compilation Example (Sheet 1 of 2)

```
(*,                                                                    *)
PROCEDURE READANDPRINT;                                                0000010
(*---------------------------------------------------------------------0000020
  PROCEDURE READANDPRINT;                                              0000030
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT. 0000040
  PROCEDURES CALLED : GETLINE, PRINTLABEL                              0000050
---------------------------------------------------------------------*) 0000060
TYPE                                                                   0000070
    LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                   0000080
VAR                                                                    0000090
    LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                   0000100
PROCEDURE GETLINE(VAR THISLINE : INTEGER); FORWARD;                    0000110
PROCEDURE PRINTLABEL; FORWARD;                                         0000120
(*+                                                                   *)
PROCEDURE PRINTLABEL;                                                  0000010
(*---------------------------------------------------------------------0000020
  PROCEDURE PRINTLABEL;                                                0000030
  PURPOSE :  PRINTLABEL PRINTS ONE COPY OF THE LABEL.                  0000040
  INPUTS :   LINESPERLABEL - NUMBER OF LINES PER LABEL                 0000050
             CHARSPERLINE - NUMBER OF CHARACTERS PER LINE              0000060
             LABELIMAGE - THE LABEL TO BE PRINTED                      0000070
---------------------------------------------------------------------*) 0000080
BEGIN                                                 (*PRINTLABEL*)   0000090
  FOR L := 1 TO LINESPERLABEL DO BEGIN                                 0000100
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );     0000110
    WRITELN; END;                                                      0000120
END;                                                  (*PRINTLABEL*)   0000130
(*-                                                                   *)
BEGIN (*$NULLBODY *)
END   (* READANDP *);
(*-                                                                   *)
BEGIN (*$NULLBODY *)
END   (* LABELS   *).
```

Figure 6-6. Source Listing, Partial Compilation Example (Sheet 2 of 2)

Figure 6-7 shows the contents of file OUTPUT for the deferred processing. The same deferred commands are used as for full compilation, and the object modules that the compiler writes are listed. The newly compiled modules for the specified routines (INTERACT and PRINTLABEL) replace the previously compiled modules as members of library ALTOBJ.

```
DXPSCLCP      1.8.0  83.357   TI 990 CONFIGURATION PROCESSOR 12/28/83     13:48:36
*USE PROCESS #
*SPLIT OBJECT
INTERACT = <ALTOBJ  ,INTERACT>:    INTERACT  12/28/83  15:02:22     DXPSCL
PRINTLAB = <ALTOBJ  ,PRINTLAB>:    PRINTLAB  12/28/83  15:02:26     DXPSCL
*EXIT
```

Figure 6-7. Contents of OUTPUT, Deferred Processing, Partial Completion

6.3.6 Source Listing

Using the following commands, CONFIG provides a listing of the source library modules specified in a process configuration:

- LIST — Specifies listing of one or more complete source modules
- LISTDOC — Lists the documentation section of one or more source modules
- LISTORDER — Specifies the listing order for the LIST and LISTDOC commands

The documentation section of a source module consists of one or more comments at the beginning of the declaration section, preceding the TYPE or VAR declaration(s). The brace ({) or parenthesis and asterisk ((*) that begin the comment must be in character position 1 or character position 1 and 2, respectively. The closing brace (}) or asterisk and parenthesis (*)) must be in character position 72 or character positions 71 and 72 of the same or of a subsequent line. The declaration section may consist of a multiline comment or of a group of comments.

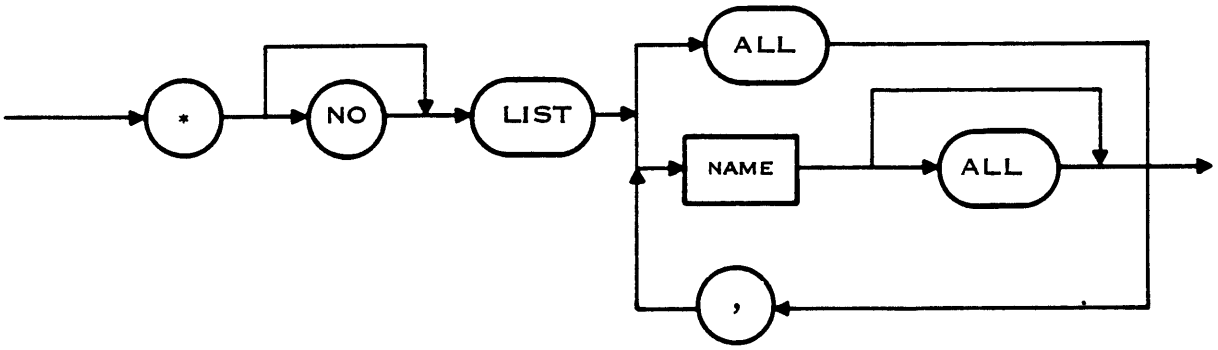
The listings are written after all commands in the file INPUT have been processed.

6.3.6.1 LIST Command. The LIST command causes CONFIG to list one or more source modules specified in the current process configuration. The syntax for the command is as follows:

(<list command> ::= *[NO] LIST ALL|*[NO] LIST<name>[ALL]{,<name>[ALL]})

The syntax diagram is as follows:

LIST Command:



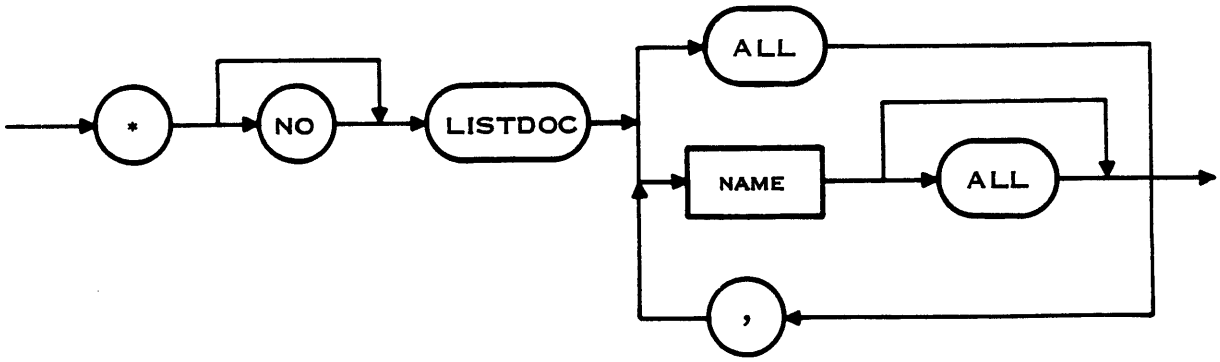
The name parameter is the name of a node in the current process configuration. The source module corresponding to each named node is listed. When the keyword ALL is entered alone, all source modules of the program are listed. When the keyword ALL is entered following a name parameter, the command lists the specified module and all descendents.

6.3.6.2 LISTDOC Command. The LISTDOC command causes CONFIG to list the documentation section of one or more source modules specified in the current process configuration. The syntax for the command is as follows:

```
<listdoc command> ::= *[NO] LISTDOC ALL
                    | *[NO] LISTDOC <name>[ALL] ,<name>[ALL]
```

The syntax diagram is as follows:

LISTDOC Command:



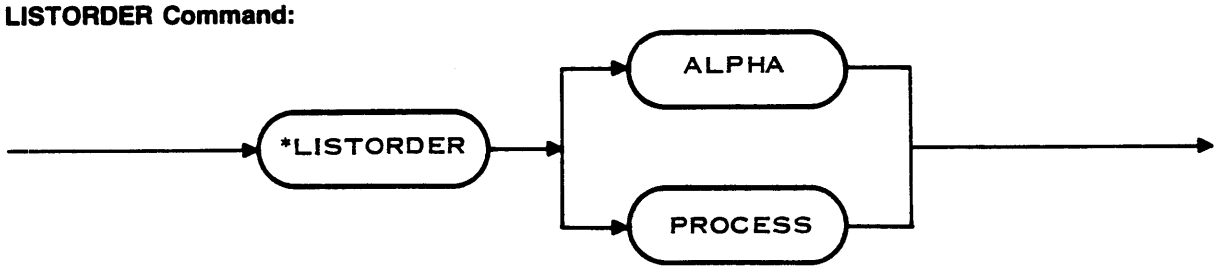
The name parameter is the name of a node in the current process configuration. The documentation section of the source module corresponding to each named node is listed. When the keyword ALL is entered alone, the documentation sections of all source modules of the program are listed. When the keyword ALL is entered following a name parameter, the command lists the documentation sections of the specified module and all descendents.

6.3.6.3 LISTORDER Command. The LISTORDER command specifies the listing order for the LIST and LISTDOC commands. The syntax of the command is as follows:

```
<listorder command> ::= *LISTORDER ALPHA | *LISTORDER PROCESS
```

The syntax diagram is as follows:

LISTORDER Command:



The keyword ALPHA causes the source modules and/or documentation sections to be listed in alphabetical order by node name. The keyword PROCESS specifies listing the source modules in the order in which they appear in the process configuration. The following is an example:

```
*LISTORDER ALPHA
```

The example command specifies listing source modules and documentation sections of source modules in alphabetic order by node name. The LISTORDER command applies to all LIST and LISTDOC commands.

6.3.6.4 Listing Examples. Figure 6-8 lists the contents of the file OUTPUT for a listing example. The commands shown are a USE PROCESS command that specifies a previously cataloged process configuration, and a LIST command that lists the entire program. The tabular representation of the process configuration is the same as for the preceding examples.

```
DXPSCLCP      1.8.0  83.357     TI 990 CONFIGURATION PROCESSOR    12/28/83 14:00:59
*USE PROCESS <LIBRARY, PROCES>
*LIST LABELS ALL

PROCESS NAME          SOURCE LOCATION       OBJECT LOCATION       FLAGS SET
-------------------------------------------------------------------------------
LABELS                <LIBRARY ,LABELS>                           2
   INTERACT           <LIBRARY ,INTERACT>                         2
   READANDP           <LIBRARY ,READANDP>                         2
      GETLINE         <LIBRARY ,GETLINE >                         2
      PRINTLAB        <LIBRARY ,PRINTLAB>                         2

INPUT                 = ST16
CRTFIL                = ST16
OUTPUT                = SYS2.DP0020.LIST
COMPFILE              = .COMPFI16
CPTEMP                = .CPTEMP16
OBJECT                = .OBJECT16

MASTER                = .MASTER16
LIBRARY               = SYS2.DP0020.SRCLIB
OBJLIB                = SYS2.DP0020.OBJLIB
ALTOBJ                = SYS2.DP0020.OBJLIB

CONFIGURATION PROCESSOR                    12/28/83                  14:01:18
LABELS    = SYS2.DP0020.SRCLIB(LABELS   )

PROGRAM LABELS;                                                              0000010
(*-------------------------------------------------------------------------- 0000020
   PROGRAM LABELS:                                                           0000030
   PURPOSE :      THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE    0000040
                  COPIES OF THAT LABEL.                                      0000050
   FILES USED:    INPUT - FOR USER-SUPPLIED PARAMETERS AND THE LABEL         0000060
                  CPTFIL - USED FOR PROMPTING INPUT                          0000070
                  OUTPUT - MULTIPLE COPIES OF THE LABEL                      0000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                                0000090
-------------------------------------------------------------------------*) 0000100
VAR CRTFIL : TEXT ;                              (*USED TO PROMPT INPUT*)    0000110
    CHARSPERLINE : INTEGER;                (*NUMBER OF CHARACTERS PER LINE*) 0000120
    LINESPERLABEL : INTEGER;                   (*NUMBER OF LINES PER LABEL*) 0000130
    COPYCOUNT : INTEGER;                       (*NUMBER OF COPIES TO PRINT*) 0000140
                                                                             0000150
PROCEDURE INTERACT; FORWARD;                                                 0000160
PROCEDURE READANDPRINT; FORWARD;                                             0000170
BEGIN                                                (*LABELS*)              0000180
   INTERACT; READANDPRINT;                                                   0000190
END.                                                 (*LABELS*)
```

Figure 6-8. Contents of File OUTPUT for LIST Command (Sheet 1 of 3)

```
CONFIGURATION PROCESSOR                    12/28/83                    14:01:18
INTERACT = SYS2.DP0020.SRCLIB(INTERACT)

PROCEDURE INTERACT;                                                     0000010
(*----------------------------------------------------------------------0000020
   PROCEDURE INTERACT;                                                  0000030
   PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.      0000040
   OUTPUTS : CHARASPERLINE - NUMBER OF CHARACTERS PER LINE              0000050
             LINESPERLABEL - NUMBER OF LINES PER LABEL                  0000060
             COPYCOUNT - NUMBER OF LABELS TO PRINT                      0000070
----------------------------------------------------------------------*) 0000080
BEGIN                                                  (*INTERACT*)     0000090
  REWRITE( CRTFIL );                                                    0000100
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                   0000110
  RESET(INPUT); READ( CHARSPERLINE );                                   0000120
  WRITELN(CRTFIL, 'HOW MANY LINES PER LABEL?' );                        0000130
  READLN; READ( LINESPERLABEL );                                        0000140
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                0000150
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');    0000160
END;                                                  (*INTERACT*)      0000170

CONFIGURATION PROCESSOR                    12/28/83                    14:01:18
READANDP = SYS2.DP0020.SRCLIB(READANDP)

PROCEDURE READANDPRINT;                                                 0000010
(*----------------------------------------------------------------------0000020
   PROCEDURE READANDPRINT;                                              0000030
   PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT. 0000040
   PROCEDURES CALLED : GETLINE, PRINTLABEL                              0000050
----------------------------------------------------------------------*) 0000060
TYPE                                                                    0000070
      LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                  0000080
VAR                                                                     0000090
      LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                  0000100
PROCEDURE GETLINE; FORWARD;                                             0000110
PROCEDURE PRINTLABEL; FORWARD;                                          0000120
BEGIN                                              (*READANDPRINT*)     0000130
   FOR L := 1 TO LINSPERLABEL DO GETLINE( LABELIMAGE(.L.) );            0000140
   FOR K := 1 TO COPYCOUNT DO PRINTLABEL;                               0000150
END;                                               (*READANDPRINT*)     0000160
```

Figure 6-8. Contents of File OUTPUT for LIST Command (Sheet 2 of 3)

```
CONFIGURATION PROCESSOR                    12/28/83                    14:01:19
GETLINE = SYS2.DP0020.SRCLIB(GETLINE )

PROCEDURE GETLINE (*VAR THISLINE : LINE*);                                0000010
(*-------------------------------------------------------------------------0000020
   PROCEDURE GETLINE;                                                     0000030
   PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.                      0000040
   INPUTS :  CHARSPERLINE NUMBER OF CHARACTERS PER LINE                   0000050
   OUTPUTS : THISLINE - THE LINE THAT WAS READ.                           0000060
-------------------------------------------------------------------------*) 0000070
VAR CH : INTEGER;                                                         0000080
BEGIN                                                      (*GETLINE*)    0000090
  READLN; CH := 1;                                                        0000100
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN                   0000110
    READ( THISLINE(.CH.) ); CH := CH + 1;                                 0000120
    END;                          (*FILL IN REST OF LINE WITH BLANKS*)    0000130
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := '';                     0000140
END;                                                       (*GETLINE*)    0000150

CONFIGURATION PROCESSOR12/28/83                                           14:01:19
PRINTLAB = SYS2.DP0020.SRCLIB(PRINTLAB) PROCEDURE PRINTLABEL;             0000010

(*-------------------------------------------------------------------------0000020
   PROCEDURE PRINTLABEL;                                                  0000030
   PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.                     0000040
   INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL                    0000050
             CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                 0000060
             LABELIMAGE - THE LABEL TO BE PRINTED                         0000070
-------------------------------------------------------------------------*) 0000080
BEGIN                                                   (*PRINTLABEL*)    0000090
   FOR L := 1 TO LINESPERLABEL DO BEGIN                                   0000100
      FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );      0000110
      WRITELN; END;                                                       0000120
END;                                                    (*PRINTLABEL*)    0000130
```

Figure 6-8. Contents of File OUTPUT for LIST Command (Sheet 3 of 3)

The source modules are listed in the order in which they appear in the process configuration. Notice that each module contains only the declarations and statements of the applicable program or routine. Only the module LABELS could be compiled alone. The other modules would fail because they do not start with a PROGRAM heading and do not end with a period (.). The modules form a source library from which CONFIG can write a source module to compile one or more of the modules of the program.

Figure 6-9 lists the contents of the file OUTPUT for an example of listing the documentation sections of a program. The commands shown are a USE PROCESS command, specifying a previously cataloged process configuration, and a LISTDOC command that lists the entire program.

```
DXPSCLCP      1.8.0  83.357     TI 990 CONFIGURATION PROCESSOR    12/28/83 14:05:21
*USE PROCESS <LIBRARY, PROCESS>
*LIST LABELS ALL

PROCESS NAME          SOURCE LOCATION       OBJECT LOCATION       FLAGS SET
-------------------------------------------------------------------------------
LABELS                <LIBRARY ,LABELS                            3
   INTERACT           <LIBRARY ,INTERACT>                         3
   READANDP           <LIBRARY ,READANDP>                         3
      GETLINE         <LIBRARY ,GETLINE >                         3
      PRINTLAB        <LIBRARY ,PRINTLAB>                         3

INPUT                 = ST16
CRTFIL                = ST16
OUTPUT                = SYS2.DP0020.LIST
COMPFILE              = .COMPFI16
CPTEMP                = .CPTEMP16
OBJECT                = .OBJECT16

MASTER                = .MASTER16
LIBRARY               = SYS2.DP0020.SRCLIB
OBJLIB                = SYS2.DP0020.OBJLIB
ALTOBJ                = SYS2.DP0020.OBJLIB

LABELS                = SYS2.DP0020.SRCLIB(LABELS  )
(*-------------------------------------------------------------------- 0000020
   PROGRAM LABELS:                                                    0000030
   PURPOSEC:      THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE 0000040
                  COPIES OF THAT LABEL.                               0000050
   FILES USED:    INPUT - FOR USER-SUPPLIED PARAMETERS AND THE LABEL  0000060
                  CPTFIL - USED FOR PROMPTING INPUT                   0000070
                  OUTPUT - MULTIPLE COPIES OF THE LABEL               0000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                         0000090
--------------------------------------------------------------------*) 0000100

INTERACT = SYS2.DP0020.SRCLIB(INTERACT)
(*-------------------------------------------------------------------- 0000020
  PROCEDURE INTERACT;                                                 0000030
  PURPOSE :  INTERACT PROMPTS THE USER, REQUESTING CERTAIN IMPUTS.    0000040
  OUTPUTS :  CHARASPERLINE - NUMBER OF CHARACTERS PER LINE            0000050
             LINEPERLABEL - NUMBER OF LINES PER LABEL                 0000060
             COPYCOUNT - NUMBER OF LABELS TO PRINT                    0000070
--------------------------------------------------------------------*) 0000080
```

Figure 6-9. Contents of File OUTPUT for LISTDOC Command (Sheet 1 of 2)

```
READANDP = SYS2.DP0020.SRCLIB(READANDP)
(*-------------------------------------------------------------------- 0000020
   PROCEDURE READANDPRINT;                                            0000030
   PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT. 0000040
   PROCEDURES CALLED : GETLINE, PRINTLABEL0000050
--------------------------------------------------------------------*) 0000060

GETLINE = SYS2.DP0020.SRCLIB(GETLINE )
(*-------------------------------------------------------------------- 0000020
   PROCEDURE GETLINE;                                                 0000030
   PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.  0000040
   INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE             0000050
   OUTPUTS : THISLINE - THE LINE THAT WAS READ.                       0000060
--------------------------------------------------------------------*) 0000070

PRINTLAB = SYS2.DP0020.SRCLIB(PRINTLAB)
(*-------------------------------------------------------------------- 0000020
   PROCEDURE PRINTLABEL;                                              0000030
   PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.                 0000040
   INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL                0000050
             CHARSPERLINE - NUMBER OF CHARACTERS PER LINE             0000060
             LABELIMAGE - THE LABEL TO BE PRINTED                     0000070
--------------------------------------------------------------------*) 0000080
```

Figure 6-9. Contents of File OUTPUT for LISTDOC Command (Sheet 2 of 2)

The documentation section of a source module consists of one or more comments at the beginning of the declaration section, preceding the TYPE declaration, if any, or the VAR declaration. The brace ({) or the parenthesis and asterisk ((*) that begin the comment must be in character position 1 or character positions 1 and 2, respectively. The closing brace (}) or asterisk and parenthesis (*)) must be in character position 72 or positions 71 and 72 of the same line or a subsequent line. The documentation section may consist of a multiline comment (as in the example) or of a group of comments.

6.3.7 Flags

For each node, the process configuration contains a set of flags that control the processing of the node. Each flag is either on or off. Flags are turned on and off by commands. When all commands have been processed, the states of all flags are passed to the external representation of the process configuration that follows the USE PROCESS# command in the deferred command file.

The two categories of flags are system flags and user flags. System flags are predefined and are set to an initial state when a process configuration is built or accessed. The states of system flags are not stored when the process configuration is stored. Table 6-1 lists the system flags, their significance, and their initial states. User flags are described in a subsequent paragraph.

The COMPILE command turns on the DECLARATION flag for each module for which the declarations are required in the source file being written. The command turns on both the DECLARATION and BODY flags for modules being compiled. The NO COMPILE command turns off the DECLARATION and BODY flags.

The LIST command turns on the LIST flag for modules to be listed, and the NO LIST command turns the LIST flag off for the specified module(s). Similarly, the LISTDOC command turns on the LISTDOC flag and the NO LISTDOC command turns the LISTDOC flag off.

The use of the CHANGED flag, set by the EDIT command, is described in a subsequent paragraph. The NEST flag may not be set or cleared by you.

The COLLECT flag is turned on and off by the Flag command (described in paragraph 6.3.7.2). When the flag is turned on for a node, that node is written to the file OBJECT during the deferred processing run. The COLLECT flag implements optional output to a file to be specified in an INCLUDE command in the link edit control file; as a result, the Link Editor need not search the library for the module.

When the COLLECT flag is set for any node of the process configuration, CONFIG places the *COLLECT OBJECT command in the deferred file following the *SPLIT OBJECT command.

The *COLLECT OBJECT command is executed during the deferred processing run when the command is read. When a USE OBJECT command or a preceding collect operation has not specified the location of the object module, CONFIG searches the library specified by ALTOBJ for the module. When the module is not on the ALTOBJ library, CONFIG next searches the library specified by OBJLIB.

The SPLIT flag is turned on and off by the Flag command. The flag is initially on, causing all modules to be cataloged as members of the specified object library. A module for which the SPLIT flag is not set is not cataloged during the deferred processing run.

The CHECK flag is turned on and off by the Flag command. The flag is initially on, causing CONFIG to check the IDT (the module identifier) on all object modules. When the CHECK flag is on, the IDT of the module is compared to the name of the node. Processing is terminated and an error message is written when the name and IDT are not identical. The IDT of the module is not checked when the CHECK flag for the node has been turned off.

You may define up to 21 user flags by using the SETFLAG command. User flags are turned on and off by the Flag command. The states of user flags are stored when the process configuration is stored. Use the Conditional Flag command to test user flags and set system flags.

Table 6-1. System Flags

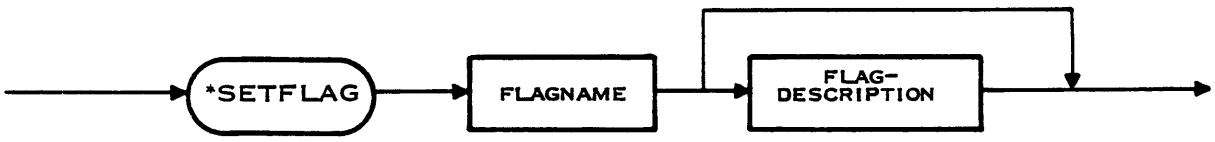
Number	Name	Description	Initial Value
0	DECLARATION	Set when declarations of this module are required in the source file	OFF
1	BODY	Set when statements of this module are required in the source file	OFF
2	LIST	Set when the source module is to be listed	OFF
3	LISTDOC	Set when the documentation section of this module is to be listed	OFF
4	CHANGED	Set when the contents of a source module are changed by an edit operation	OFF
5	NEST	Not currently used	
6	SPLIT	Set when the object module is to be written as a member of library OBJLIB or ALTOBJ	ON
7	COLLECT	Set when the object module is to be written on the OBJECT file	OFF
8	CHECK	Set when the IDT of the module is to be compared to the name of the node	ON

6.3.7.1 SETFLAG Command. The SETFLAG command defines or deletes the definition of a user flag. The syntax of the command is as follows:

<setflag command> ::= *SETFLAG <flagname> [<flag-description>]

The syntax diagram is as follows:

SETFLAG Command:



The parameter flagname consists of one to eight characters and may not be a CONFIG keyword. The flag description is a string of up to 64 characters that describes the flag. The flag description begins with the first nonblank character following the flagname parameter and extends to the first asterisk (*), normally the asterisk that begins the next command. The flag description may contain blanks and serves as a comment to identify the flag. When the flag description is omitted, the definition of that flag is deleted and the flag is turned off in all nodes in the program.

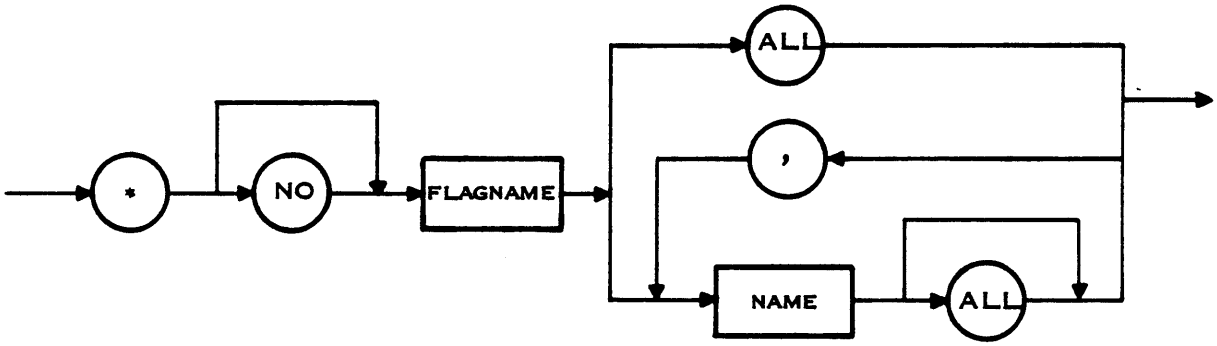
6.3.7.2 Flag Command. The Flag command turns certain system flags and all user flags on and off. The syntax for the command is as follows:

```
<flag command> ::= *[NO] <flagname>ALL
                 | *[NO] <flagname> <name> [ALL] {,<name> [ALL]}

<flagname>::= SPLIT | COLLECT | CHECK | <user flagname>
```

The syntax diagram is as follows:

Flag Command:



The flag is turned off when the optional keyword NO is entered. Otherwise, the flag is turned on. When the keyword ALL immediately follows the flagname parameter, the flag is turned on or off in all nodes of the current process configuration. The name parameter specifies a node in which the flag is turned on or off. When the name parameter is followed by the optional keyword ALL, the FLAG is turned on or off in the named node and in all of its descendents.

6.3.7.3 Conditional Flag Command. The Conditional Flag command tests a specified flag and turns another specified flag on or off according to the result. The syntax of the command is as follows:

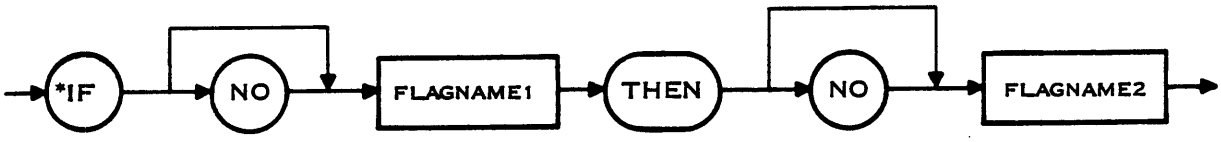
```
<conditional flag command> :: = *IF [NO] <flagname1>
                                THEN [NO] <flagname2>

<flagname1> :: = COMPILE | LIST | LISTDOC | CHANGED | SPLIT
               | COLLECT | CHECK | <user flagname>

<flagname2> ::= COMPILE |LIST | LISTDOC | SPLIT | COLLECT
               | CHECK | <user flagname>
```

The syntax diagram is as follows:

Conditional Flag Command:



The flagname1 parameter specifies a flag to be tested in all nodes of the current process configuration. The optional keyword NO preceding filename1 specifies the state. When NO is entered, the flag is tested for the off state. Otherwise, the flag is tested for the on state. In each node for which the test is successful, flagname2 is turned on if the optional keyword NO is omitted or off if NO is entered.

Notice that COMPILE is allowed as a flagname parameter in the Conditional Flag command, even though COMPILE is not the name of a flag. When COMPILE is entered as flagname1, the BODY flag is tested. When COMPILE is entered as flagname2, the BODY flag is turned on or off as specified. When the BODY flag is turned on, the DECLARATION flag is also turned on.

6.3.7.4 Flag Examples. The Flag commands allow you to control program processing by setting or resetting the system flags. The following is an example of a FLAG command:

```
*COLLECT INTERACT
```

The command turns on the COLLECT flag for module INTERACT and causes CONFIG to include the *COLLECT OBJECT in the deferred command file following the *SPLIT OBJECT command.

The deferred processing run of CONFIG writes the module for INTERACT (and any others for which the COLLECT flag is on) to the OBJECT file. The OBJECT file can be specified in an INCLUDE command to the Link Editor.

Another example of a Flag command is as follows:

```
*NO SPLIT ALL
```

This command turns off the SPLIT flags for all modules. The deferred processing run of CONFIG does not catalog the object modules. Unless the COLLECT flag is set for one or more modules, the use of this example is of little value.

The Conditional Flag command allows you to turn the system flags on and off selectively. The following is an example of a Conditional Flag command:

```
*IF CHANGED THEN COMPILE
```

The EDIT command turns on the CHANGED flag when the module is edited. The command in the example also turns on the DECLARATION and BODY flags for those modules if they are off.

User flags may be defined to support an overlay structure, allowing you to specify processing for overlays. The following are examples of the use of the SETFLAG command to define user flags:

```
*SETFLAG OVRLAY1 OVERLAY 1 MODULE
*SETFLAG OVRLAY2 OVERLAY 2 MODULE
```

FLAG commands turn on the flags in the specified modules:

```
*OVRLAY1 READIN ALL
*OVRLAY2 PRINT ALL
```

As a result of these commands, user flag OVRLAY1 is turned on in module READIN and in its descendents and user flag OVRLAY2 is turned on in module PRINT and in its descendents. The command in the following example causes CONFIG to collect overlay 2 modules in file OBJECT:

```
*IF OVRLAY2 THEN COLLECT
```

6.3.8 Modifying a Process Configuration

The examples in this section show how to build a process configuration and use it for additional processing. However, you can modify the current process configuration in several ways. Using ADD commands you can add modules to the program structure defined in the process configuration. You can use DELETE commands to delete nodes and MOVE commands to modify the structure by moving nodes to other points. Using the DISPLAY command, you can display the process configuration. You can use the USE OBJECT command to specify object locations for nodes and USE commands to specify or change source locations. Use the DEFAULT SOURCE and DEFAULT OBJECT commands to change default libraries for source and object modules.

6.3.8.1 DELETE Command. The DELETE command deletes a module and its descendents, if any, from the current process configuration. The syntax of the command is as follows:

```
<delete command> ::= *DELETE <name>
```

The syntax diagram is as follows:

DELETE Command:

The name parameter is the name of the node to be deleted. When the named node has descendents, the descendents are also deleted. The following is an example:

```
*DELETE READANDP
```

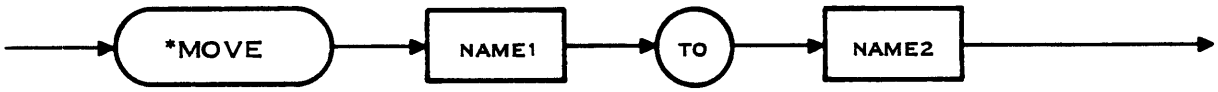
The example command deletes node READANDP and its descendants, GETLINE and PRINTLAB. It can be followed by one or more ADD commands to change the names of these nodes of the process configuration.

6.3.8.2 MOVE Command. The MOVE command moves a module and all of its descendents to become a son of another module. The syntax of the command is as follows:

<move command> ::= *MOVE <name1> TO <name2>

The syntax diagram is as follows:

MOVE Command:



The name1 parameter is the name of a node to be moved. The name2 parameter is the name of another node in the structure. The parameter name2 cannot be a descendent of name1. The node specified as name1 and all its descendents, if any, are moved. The node specified as name1 becomes a son of the node specified as name2. The MOVE command implies changes in the declarations of routines within the source code. An example is as follows:

*MOVE GETLINE TO LABELS

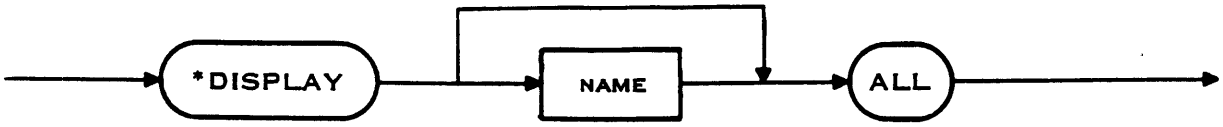
The example command moves node GETLINE (a son of READANDP) to become a son of LABELS. Text edit the files to move the forward declaration of GETLINE in the declaration portion of READANDP to the declaration portion of LABELS to compile any module correctly.

6.3.8.3 DISPLAY Command. The DISPLAY command displays the tabular representation of all or part of the current process configuration on file CRTFILE. The syntax is as follows:

<display command> ::= *DISPLAY [<name>] ALL

The syntax diagram is as follows:

DISPLAY Command:



When the optional name parameter is omitted, the entire current process configuration appears on the screen. When the name parameter is included, the name is the name of a node, and the portion of the process configuration that lists the named node and its descendents appears on the screen.

The DISPLAY command allows you to display the process configuration, noting the effects of the commands that have been processed. The following is an example:

*DISPLAY READANDP ALL

The resulting display of the current process configuration includes nodes READANDP, GETLINE, and PRINTLAB.

The format of the display is similar to that written to file OUTPUT (Figure 6-9). The names of the nodes are displayed, indented to show the program structure. The source and object locations are displayed, as are the numbers of system flags that are not in their initial states. The initial states of flags 6 and 8 (SPLIT and CHECK) are on; the initial states of the other system flags are off. Except for flags 6 and 8, the display of the flag number indicates that it is on. An asterisk (*) appears with numbers 6 and 8 because the display of either of these numbers means that the flag is off.

6.3.8.4 USE OBJECT Command. The USE OBJECT command specifies a location for the object module of a specified node. The syntax of the command is as follows:

<use object command> ::= *USE OBJECT <name> <location>

The syntax diagram is as follows:

USE OBJECT Command:

The name parameter is the name of the node to which the object location applies. The location parameter syntax is as follows:

<location> ::= "<"[<library>,]<member>">"

When the library is not specified in the location parameter, the location for the object module remains unspecified. The following is an example:

*USE OBJECT INTERACT <OBJL1,INTERACT>

The example specifies that the object module for node INTERACT is to be cataloged as member INTERACT of the library whose synonym is OBJL1.

When a USE OBJECT command specifies a member name or a library synonym and a member name, CONFIG writes the object module to the object library member when it performs the deferred processing. Otherwise, CONFIG writes the module to the default object library, using the node name as the member name. The representation of the process configuration does not contain an object location unless a USE OBJECT or DEFAULT OBJECT command has been entered.

6.3.8.5 USE Command. The USE command specifies a location for the source module for a specified node. The syntax of the command is as follows:

<use command> ::= *USE <name> <location>

The syntax diagram is as follows:

USE Command:

The name parameter is the name of the node to which the source location applies. The location parameter syntax is as follows:

<location> ::= "<"[<library>,]<member>">"

The following is an example:

*USE GETLINE <SLIB1,INPLIN>

The example specifies that the source module for node GETLINE is to be cataloged as member INPLIN of the library whose synonym is SLIB1.

You can use either a USE command or an ADD command to specify a source library synonym and member name for a source module. To assign a different library synonym or member name, use the USE command.

6.3.9 Libraries

The following library synonyms are initially defined in CONFIG:

- MASTER — Intended for source modules of tested (fully developed) programs
- OBJLIB — Intended for object modules corresponding to source modules in MASTER
- LIBRARY — Intended for source modules of programs under development
- ALTOBJ — Intended for object modules corresponding to source modules in LIBRARY

The default source library synonym LIBRARY is the logical default because it is intended for programs under development. Similarly, the default object library synonym ALTOBJ is appropriate because it is intended for object modules corresponding to the source modules in LIBRARY. To change either default value use the DEFAULT SOURCE or DEFAULT OBJECT commands.

CONFIG maintains a library table in the process configuration. The first four entries in the table are the initially defined library synonyms MASTER, LIBRARY, OBJLIB, and ALTOBJ. When a library synonym is entered in any of the commands that may include a library synonym parameter, the synonym is added to the library table (unless it already appears in the table). The commands are BUILD PROCESS, CAT PROCESS, USE PROCESS, ADD, USE, USE OBJECT, DEFAULT SOURCE, DEFAULT OBJECT, and EDIT.

Other synonyms may be substituted for the initially defined library synonyms. The MASTER command specifies a library synonym to replace MASTER. Similarly, the LIBRARY, OBJLIB, and ALTOBJ commands specify library synonyms to replace LIBRARY, OBJLIB, and ALTOBJ, respectively.

A library is identified to CONFIG using a library synonym, the value of which is the pathname of a library file. Assign a synonym by using the Assign Synonym (AS) SCI command prior to executing CONFIG. Alternately, you can assign a synonym by using a SETLIB command.

6.3.9.1 MASTER Command. The MASTER command specifies a library synonym to replace the initially defined library synonym MASTER. The syntax of the command is as follows:

<master command> ::= *MASTER <libname>

The syntax diagram is as follows:

MASTER Command:

The libname parameter is a library synonym that replaces synonym MASTER as the first entry in the library table. When the MASTER command is used, it should precede the ADD commands that define the nodes of the process configuration. The following is an example:

*MASTER SRCLIB1

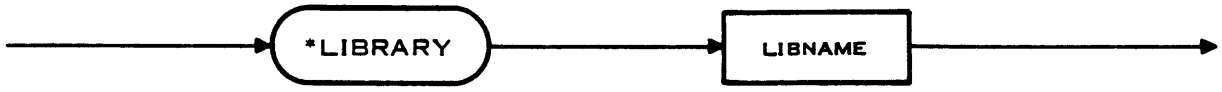
In this example, SRCLIB1 replaces MASTER as the library synonym in the first entry of the library table.

6.3.9.2 LIBRARY Command. The LIBRARY command specifies a library synonym to replace the initially defined library synonym LIBRARY. The syntax of the command is as follows:

<library command> ::= *LIBRARY <libname>

The syntax diagram is as follows:

LIBRARY Command:



The libname parameter is a library synonym that replaces synonym LIBRARY as the second entry in the library table. When the LIBRARY command is used, it should precede the ADD commands that define the nodes of the process configuration. The following is an example:

*LIBRARY SRCLIB2

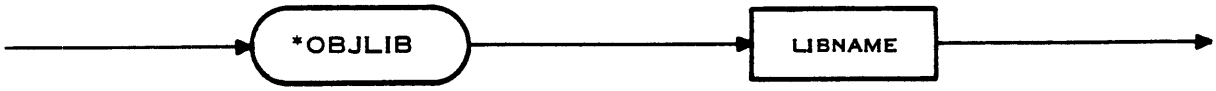
In this example, SRCLIB2 replaces LIBRARY as the library synonym in the second entry of the library table.

6.3.9.3 OBJLIB Command. The OBJLIB command specifies a library synonym to replace the initially defined library synonym OBJLIB. The syntax of the command is as follows:

<objlib command> ::= *OBJLIB <libname>

The syntax diagram is as follows:

OBJLIB Command:



The libname parameter is a library synonym that replaces synonym OBJLIB as the third entry in the library table. When the OBJLIB command is used, it should precede the ADD commands that define the nodes of the process configuration. The following is an example:

*OBJLIB OBJLIB1

In this example, OBJLIB1 replaces OBJLIB as the library synonym in the third entry of the library table.

6.3.9.4 ALTOBJ Command. The ALTOBJ command specifies a library synonym to replace the initially defined library synonym ALTOBJ. The syntax of the command is as follows:

<altobj command> ::= *ALTOBJ <libname>

The syntax diagram is as follows:

ALTOBJ Command:

The libname pathname is a library synonym that replaces synonym ALTOBJ as the fourth entry in the library table. When the ALTOBJ command is used, it should precede the ADD commands that define the nodes of the process configuration. The following is an example:

*ALTOBJ OBJLIB2

In this example, OBJLIB2 replaces ALTOBJ as the library synonym in the fourth entry of the library table.

6.3.9.5 SETLIB Command. The SETLIB command assigns a value to a synonym. The syntax of the command is as follows:

<setlib command> ::= *SETLIB <libname> <value>

The syntax diagram is as follows:

SETLIB Command:

The libname parameter is a library synonym that meets the requirements for a TIP identifier. A TIP identifier includes a maximum of 72 alphanumeric, dollar sign ($), or underscore (_) characters, beginning with a letter or $. However, CONFIG uses only the first eight characters. The value is the pathname of the library file to which the synonym applies. The following is an example:

*SETLIB SRCLIB3 DS02.PASCAL.SOURCE.GARY

This example command accesses the operating system to assign the value DS02.PASCAL.SOURCE.GARY to SRCLIB3.

Use the SETLIB command with caution to avoid defining too many synonyms for the library file pathname.

6.3.9.6 DEFAULT SOURCE Command. The DEFAULT SOURCE command specifies the library synonym for the default source library. The synonym applies to modules defined by subsequent ADD commands. The syntax of the command is as follows:

<default source command> ::= *DEFAULT SOURCE <libname>

The syntax diagram is as follows:

DEFAULT SOURCE Command:

The libname parameter is the synonym that has the pathname of the library as its value. Synonym MASTER is the initially defined alternate source library synonym; however any library synonym is acceptable. The DEFAULT SOURCE command does not alter the stored process configuration; it applies only to the current run. The following is an example:

*DEFAULT SOURCE SLIB1

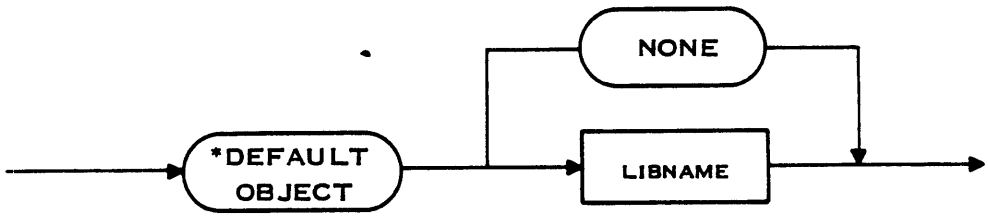
The example command specifies library synonym SLIB1 as the default source library. Source locations that do not explicitly include a library synonym use SLIB1 as the library synonym until the default value is changed by another DEFAULT SOURCE command. The default source library synonym is LIBRARY until the first DEFAULT SOURCE command is entered.

6.3.9.7 DEFAULT OBJECT Command. The DEFAULT OBJECT command specifies the library synonym for the default object library. The syntax of the command is as follows:

```
<default object command> ::= *DEFAULT OBJECT <libname>
                           | *DEFAULT OBJECT NONE
```

The syntax diagram is as follows:

DEFAULT OBJECT Command:



The libname parameter is the synonym that has the pathname of the library as its value. Synonym OBJLIB is the initially defined alternate object library synonym; however, any library synonym is acceptable.

Entering keyword NONE instead of a libname parameter restores the initial default. The DEFAULT OBJECT command does not alter the stored process configuration; it applies only to the current run.

Prior to the entry of a DEFAULT OBJECT command and subsequent to the entry of a DEFAULT OBJECT NONE command, no object locations appear in the representation of the process configuration on the OUTPUT file; also, ALTOBJ is the default object library on which object modules are stored. The following is an example:

```
*DEFAULT OBJECT OBJL1
```

The example command specifies library synonym OBJL1 as the default object library. Object locations that do not explicitly include a library synonym use OBJL1 as the library synonym until another DEFAULT OBJECT command changes the default value.

6.3.10 Text Editing

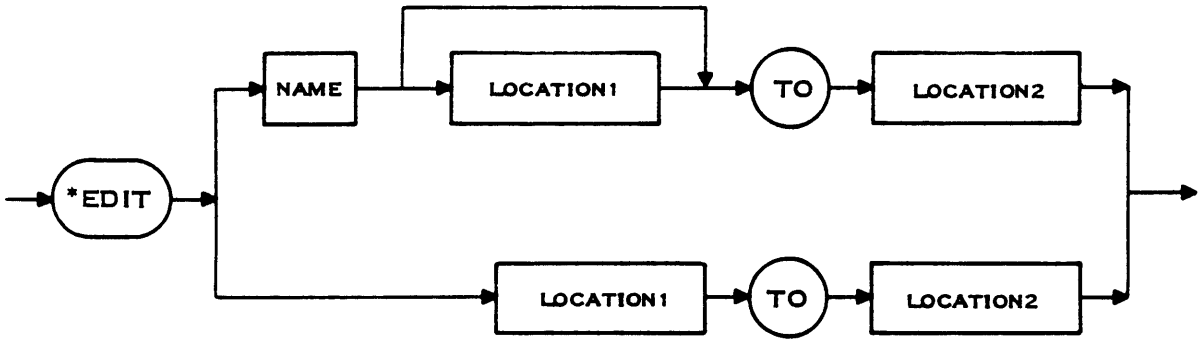
CONFIG provides a line-oriented text editing capability for editing source modules. The modules need not apply to a node or nodes of the current process configuration. The EDIT command specifies the source module to be edited and copies the file, with specified alterations, to another location. The Insert command inserts one or more source lines at a specified point. The Replace command replaces a specified source line (or lines) with one or more source lines.

6.3.10.1 EDIT Command. When the EDIT command specifies a source module that is a node of the current process, the CHANGED flag is turned on for that node. The syntax for the command is as follows:

<edit command> ::= *EDIT [<name>][<location1>] TO <location2>

The syntax diagram is as follows:

EDIT Command:



The name parameter is the node name corresponding to the source module to be edited. The syntax for the location parameters is as follows:

<location> ::= "<"[<library>,]<member>">"

The source module at location1 is edited, and the result is copied and cataloged at location2.

When the name parameter is used, the source module associated with the named node of the current process configuration is edited and the CHANGED flag for the node is turned on. The location for the source module of the node is changed to location2.

When the name parameter is omitted, the source module at location1 is edited. The source module may or may not be associated with the current process configuration; however, the CHANGED flag is not set in eitner case.

The following is an example:

*EDIT INTERACT TO <LIBRARY,NEWMOD>

This command specifies that the Insert and Replace commands that follow apply to the source module corresponding to node INTERACT. The edited module is cataloged as member NEWMOD of the library corresponding to library synonym LIBRARY. This module becomes the source module for node INTERACT. The source location or node INTERACT becomes LIBRARY,NEWMOD and the CHANGED flag is set for the node. The following is an example:

*EDIT<LIBRARY,ORIG> TO <SRCLIB1,NEWMOD>

This example command specifies editing the source module at location <LIBRARY,ORIG> and cataloging the resulting module at location <SRCLIB1,NEWMOD>. The source module at location LIBRARY, ORIG may or may not be a node of a process configuration; this command does not alter the flags of process configuration nodes.

6.3.10.2 Insert Command. The Insert command inserts one or more source lines in the source module identified in the preceding EDIT command. The lines to be inserted follow the command, and the command parameter specifies the line in the source module after which the new lines are to be inserted. The syntax of the command is as follows:

<insert command> ::= -<line>

The syntax diagram is as follows:

Insert Command:

The line parameter is the line number of the source module line after which the new lines are inserted. More than one Insert command may be supplied for editing a source module; these may be interspersed with Replace commands. These commands must be ordered by line number, in ascending numerical order. The following is an example:

-40

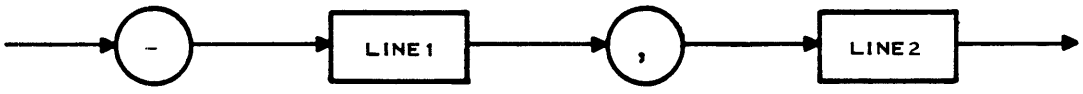
Insert the lines that follow this command after line 40 of the source module.

6.3.10.3 Replace Command. The Replace command replaces one or more source lines in the source module identified in the preceding EDIT command. The lines that replace the source module lines follow the command, and the command parameters specify the lines to be replaced. The syntax of the command is as follows:

<replace command> ::= -<line1>,<line2>

The syntax diagram is as follows:

Replace Command:



The line1 and line2 parameters are the line numbers of the source module lines to be replaced. The source lines, starting with line1 and extending through line2, are replaced by the lines that follow the Replace command. The number of replacing lines may be larger, smaller, or equal to the number of replaced lines. More than one Replace command may be entered for editing a source module; these may be interspersed with Insert commands. These commands must be ordered by line numbers, in ascending order. The following is an example:

```
-50,70
```

Replace lines 50 through 70 with the lines that follow this command.

6.3.11 Required Files

CONFIG requires seven files, listed in Table 6-2. When the files are not specified, CONFIG uses the default names. The default names consist of the file names (or the first six characters of file names longer than six characters) with the digits of your station number concatenated at the right. For example, the default name for the file supplied to the compiler is COMPFI08 when CONFIG is executed at terminal ST08.

Table 6-2. Files Required for CONFIG

Name	I/O	Description
INPUT	I	Contains CONFIG commands
OUTPUT	O	Contains CONFIG listings
SYSMSG	O	Contains system messages
COMPFILE	O	Contains source code selected by CONFIG for compilation
CRTFILE	O	Contains input commands and error messages
CPTEMP	I/O	Contains deferred processing commands
OBJECT	I/O	Contains object file from CODEGEN and object file written by CONFIG

6.3.12 Executing CONFIG

To execute the configuration processor, use either of two SCI procedures: XCONFIG or XCONFIGI. Procedure XCONFIG requests access names for all files. Procedure XCONFIGI uses your terminal for three of the files so that commands can be entered interactively. You can enter either SCI command at any time the system is ready for a command. After you enter XCONFIG, the following prompts appear:

```
EXECUTE CONFIG PROCESSOR  <VERSION: X.X.X YYDDD>

            COMMANDS: pathname@
            CRT FILE: devicename@ (DUMY)
             LISTING: [pathname@]
            MESSAGES: [pathname@] (*)
                MODE: {FOREGROUND/BACKGROUND/BATCH} (BACKGROUND)
              SOURCE: [filename@]
              OBJECT: [filename@]
              MEMORY: integer,integer  (4,8)
```

All but two of the items require access names of devices or files. Table 6-2 lists the applicable file names. The prompts request the following information:

- COMMANDS — Access name of a file (file name INPUT) that contains CONFIG commands or of a device at which commands are to be entered
- CRT FILE — Access name of a device (file name CRTFILE) to display commands and error messages
- LISTING — Access name of a device or file (file name OUTPUT) for listing
- MESSAGES — Access name of a device or file (file name SYSMSG) for system messages
- MODE — Mode of execution (foreground, background, or batch)
- SOURCE — Access name of a file (file name COMPFILE) for source file output
- OBJECT — Access name of a file (file name OBJECT) for the object file
- MEMORY — An ordered pair specifying stack and memory requirements (same as for the compiler)

When XCONFIGI is entered, the following information is requested:

```
EXECUTE CONFIG INTERACTIVELY  <VERSION: X.X.X YYDDD>

            LISTING: [pathname@]
             SOURCE: [filename@]
             OBJECT: [filename@]
             MEMORY: integer,integer (4,8)
```

The prompts COMMAND, CRT FILE, and MESSAGES are not displayed because they are assigned to ME, the synonym for your terminal. The prompt MODE is not included since the system specifies foreground. Other prompts and responses are identical to those for the XCONFIG procedure. The two procedures may be used interchangeably. When the command file, display file, and system message file are assigned to your terminal, use XCONFIGI. Otherwise, use XCONFIG.

The distinction between the run of CONFIG that prepares the source file for compilation and the run of CONFIG that splits the object file and catalogs the object modules is the command file. When the command file is the deferred processing command file written by a previous run of CONFIG (or a command file that contains those commands), the deferred processing occurs. Otherwise, the initial processing occurs.

CONFIG sets the condition code synonym $$CC to indicate the termination status as follows:

Value	Status
>0000	Normal termination
>4000	Warning condition detected
>6000	Recoverable errors detected
>8000	Fatal errors detected
>C000	Abnormal termination

NOTE

Refer to paragraphs 5.4 and 10.3.3 for further information on $$CC.

Warnings are simple user mistakes, such as invalid command syntax. Recoverable errors are errors encountered when CONFIG is executed interactively; these types of errors can be corrected by the user. Fatal errors detected by CONFIG prevents meaningful output from being produced. Any error that CONFIG cannot resolve in any way causes abnormal termination.

Procedure XTIP does not execute CODEGEN when the SILT phases detect errors. When you compile a program that consists of many modules that CONFIG will catalog as a library and when CODEGEN is executing, several of the resulting modules may be error free. Consequently, recompiling is unnecessary. Use procedures XSILT and XCODE instead of procedure XTIP to execute CODEGEN unconditionally. You can use these procedures with CONFIG.

Using a batch stream usually facilitates the execution of CONFIG. Figure 6-10 shows a sample batch stream to execute CONFIG, SILT, CODEGEN, and the deferred processing run of CONFIG. When executing this batch stream, enter a WAIT command immediately following the Execute Batch (XB) command. The CONFIG commands are entered interactively at this point. Establish a process with either a *USE PROCESS or *BUILD PROCESS command, and then enter *COMPILE ALL. Press the Return key following the last command to terminate entry of commands and press the Enter key to continue execution. The batch stream creates the required files on a user directory. Enter the actual pathname of the directory in the first .SYN command.

```
BATCH
*         SAMPLE BATCH STREAM TO PERFORM A CONFIG / COMPILE / CONFIG
*         SEQUENCE
*
*              DELETE TIP "SECRET" SYNONYMS TO RELEASE SPACE
*              IN THE SYNONYM TABLE.
*
P$SYN
*
*              ASSIGN LIBRARY SYNONYMS.
*
.SYN USER      = your directory name
.SYN MASTER    = USER.SRC
.SYN LIBRARY   = USER.SRC
.SYN OBJLIB    = USER.OBJ
.SYN ALTOBJ    = USER.OBJ
*
*              EXECUTE CONFIG INTERACTIVELY TO STAGE THE
*              TIP SOURCE FOR COMPILATION.
*
XCONFIGI       LISTING = USER.CF1
*
*              EXECUTE THE COMPILER.
*
XSILT          LISTING = USER.CM1, MESSAGES = USER.CM2
XCODE
*
*              EXECUTE THE CONFIGURATION PROCESSOR TO
*              SPLIT THE OBJECT MODULES.
*
XCONFIG        LISTING = USER.CF2, MESSAGES = USER.CF3
*
*              BUILD A SINGLE LISTING FILE.
*
CC        INPUT = (USER.CF1,USER.CM1,USER.CM2,USER.CF2,USER.CF3),
         OUTPUT = USER.LIST,
        REPLACE = YES
*
*              DELETE TEMPORARY FILES (IF DESIRED).
*
DF     PATHNAME = (USER.CF1,USER.CM1,USER.CM2,USER.CF2,USER.CF3)
EBATCH
```

Figure 6-10. Batch Stream for Separate Compilation

Notice that the synonym LIBRARY is assigned to the pathname of the user directory with .SRC concatenated at the right. This is acceptable only if one of the following conditions holds true:

- The source modules supplied as input to CONFIG have been cataloged as members of a library having that pathname.
- The location parameters in the process configuration do not use the default library synonym, LIBRARY.

Use the location parameters of the BUILD PROCESS, ADD, and CAT PROCESS commands to specify pathnames other than those supplied in the batch stream, as required. Alternately, you can modify file names to be compatible with the batch stream.

6.4 SPLIT PROGRAM UTILITY

The Split Program utility, SPLITPGM, divides a TIP program into modules and catalogs these modules as members of a library file. SPLITPGM also writes an input command file for CONFIG, which contains the commands required to build the process configuration corresponding to the original source program structure.

The program source modules should meet the format requirements for CONFIG because the normal use of the library provided by SPLITPGM is as input to CONFIG. Specifically, SPLITPGM requires the forward declarations and the indentations that CONFIG also requires. The required indentations are most easily provided by submitting the source code to Nester prior to executing SPLITPGM.

To add the forward declarations to the output of Nester, use the Text Editor. The following is a typical procedure declaration:

```
PROCEDURE GETLINE (VAR THISLINE : LINE);
```

This becomes a forward declaration when the keyword FORWARD is added, as follows:

```
PROCEDURE GETLINE (VAR THISLINE : LINE); FORWARD;
```

The declarations and statements for the procedure require a heading that does not include the parameter list. Add this as a comment to promote documentation of the program, as follows:

```
PROCEDURE GETLINE; (*VAR THISLINE : LINE*)
```

To add forward declarations for functions, the keyword FORWARD follows the parameter list and the function result type. The routine heading includes only the function name, not the parameter list and function type.

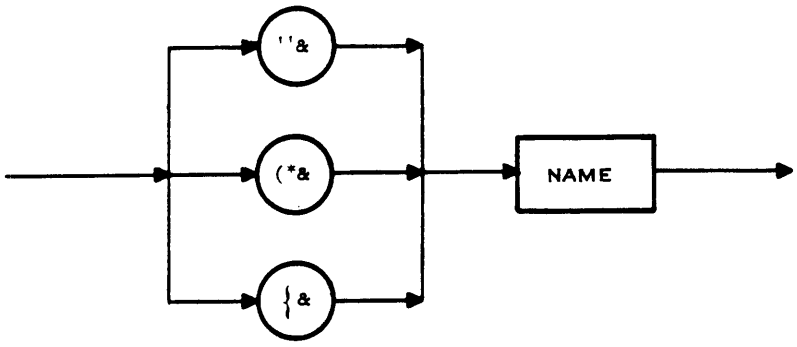
6.4.1 Split Program Command

The Split Program command specifies the node name for a source module at the beginning of each module. The syntax of the command is as follows:

```
<split program command> ::= "&<name>|(*&<name>|"{"&<name>
```

The syntax diagram is as follows:

Split Program Command:



The name parameter is the name that SPLITPGM supplies in the command file as the node name. Unless the CHECK flag for each node is turned off, CONFIG checks that the node name is identical to the IDT name of the source module. Therefore, the name parameter in the command should be the IDT of the module. All name parameters for the modules of a program should be unique in the first six characters because SPLITPGM uses the names of members of LIBRARY. Also, do not use an underscore as one of the first six characters since it is not allowed in a pathname. The Split Program command for a module must precede the heading (PROGRAM, PROCEDURE, or FUNCTION) of the module and any comments or option comments that precede the heading. The following is an example:

```
"& GETLINE
```

This command should be inserted between the forward declaration and the heading of procedure GETLINE.

6.4.2 Input Example

Figure 6-11 shows the result of changing procedure headings to forward declarations and adding procedure headings and Split Program commands in the example program. Alternately, the source file generated by the Configuration Processor is suitable input for SPLITPGM.

6.4.3 Library and Files

SPLITPGM catalogs the source modules defined by the Split Program commands as members of a library file identified when the SPLITPGM utility is invoked. Be careful when splitting source modules—existing files will be replaced in the source library.

SPLITPGM uses the following files:

File Name	I/O	Description
INPUT	I	Contains the TIP program to be split (file)
OUTPUT	O	Contains the CONFIG commands to build a process configuration (file)
LIBRARY	O	The library where individual source modules are placed (directory)
SYSMSG	O	Contains the CONFIG messages (file)

6.4.4 Execution

The SCI command XSPLITPG can be used independently from CONFIG to split source modules output by the Configuration Processor into individual files. Enter XSPLITPG, and the following prompts appear:

```
PASCAL SPLIT PROGRAM UTILITY <VERSION: X.X.X YYDDD>

      SOURCE FILE INPUT:  pathname@
   COMMAND FILE OUTPUT:  pathname@
      SOURCE DIRECTORY:  pathname@ (LIBRARY)
              MESSAGES:  [pathname@]  (*)
                  MODE:  {FOREGROUND/BACKGROUND} (BACKGROUND)
```

These prompts request the following information:

- SOURCE FILE INPUT — Access name of the file containing source modules to be split. This file can be created by the user, or can be the compilable source file generated as output by the Configuration Processor;
- COMMAND FILE OUTPUT — Access name of the file that will receive the Configuration Processor commands generated by the SPLITPGM utility. These commands can be used to build a new process record.
- SOURCE DIRECTORY — Access name of the directory where the individual source files will be placed.
- MESSAGES — Access name of the file to contain the SPLITPGM messages. As a module is split out from the source file, its name is written to the message file.
- MODE — Mode of execution, either foreground or background.

SPLITPGM divides the source program into modules as defined by the commands and the BEGIN and END keywords that are the limits of the statement portions of the modules. To list the contents of the resulting module, submit a *LIST ALL command to the configuration processor. Figure 6-8 shows the contents of each source module of the example program shown in Figure 6-11.

The following batch stream executes SPLITPGM and CONFIG to build the process configuration for the program being split:

```
BATCH
XSPLITPG SFI="<input source file>", CFO="<output file>",
    SD="<output source library>"
XCONFIG COMMANDS="<output file>",LISTING="<listing file>"
EBATCH
```

The pathname is that of the directory of the library to which the source library is to be written. The listing file is the access name of a device or file for the CONFIG listing. The input source file and the output file are as defined previously.

```
"&LABELS
PROGRAM LABELS;                                                              0000010
(*-------------------------------------------------------------------------- 0000020
   PROGRAM LABELS:                                                           0000030
   PURPOSE :     THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE     0000040
                 COPIES OF THAT LABEL.                                       0000050
   FILES USED:   INPUT - FOR USER-SUPPLIED PARAMETERS AND THE LABEL          0000060
                 CPTFIL - USED FOR PROMPTING INPUT                           0000070
                 OUTPUT - MULTIPLE COPIES OF THE LABEL                       0000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                                0000090
--------------------------------------------------------------------------*) 0000100
VAR CRTFIL : TEXT ;                                (*USED TO PROMPT INPUT*)  0000110
CHARSPERLINE : INTEGER;                  (*NUMBER OF CHARACTERS PER LINE*)   0000120
LINESPERLABEL : INTEGER;                     (*NUMBER OF LINES PER LABEL*)   0000130
COPYCOUNT : INTEGER;                         (*NUMBER OF COPIES TO PRINT*)   0000140
PROCEDURE INTERACT; FORWARD;
"&INTERACT
PROCEDURE INTERACT;                                                          0000150
(*-------------------------------------------------------------------------- 0000160
   PROCEDURE INTERACT;                                                       0000170
   PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.           0000180
   OUTPUTS : CHARASPERLINE - NUMBER OF CHARACTERS PER LINE                   0000190
             LINESPERLABEL - NUMBER OF LINES PER LABEL                       0000200
             COPYCOUNT - NUMBER OF LABELS TO PRINT                           0000210
--------------------------------------------------------------------------*) 0000220
BEGIN                                                        (*INTERACT*)    0000230
   REWRITE( CRTFIL );                                                        0000240
   WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                       0000250
   RESET(INPUT); READ( CHARSPERLINE );                                       0000260
   WRITELN(CRTFIL, 'HOW MANY LINES PER LABEL?' );                            0000270
   READLN; READ( LINESPERLABEL );                                            0000280
   WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                    0000290
   READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');        0000300
END;                                                         (*INTERACT*)    0000310
PROCEDURE READANDPRINT; FORWARD                                              0000320
"&READANDPRINT
PROCEDURE READANDPRINT;
(*-------------------------------------------------------------------------- 0000330
   PROCEDURE READANDPRINT;                                                   0000340
   PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.    0000350
   PROCEDURES CALLED : GETLINE, PRINTLABEL                                   0000360
--------------------------------------------------------------------------*) 0000370
TYPE                                                                         0000380
      LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                       0000390
VAR                                                                          0000400
      LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                       0000410
PROCEDURE GETLINE(VAR THISLINE : LINE); FORWARD;                             0000420
"&GETLINE
PROCEDURE GETLINE(*VAR THISLINE : LINE*);
```

Figure 6-11. Example of Input to SPLITPGM (Sheet 1 of 2)

```
(*-------------------------------------------------------------------------- 0000430
   PROCEDURE GETLINE;                                                       0000440
   PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.                        0000450
   INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                   0000460
   OUTPUTS : THISLINE - THE LINE THAT WAS READ.                             0000470
--------------------------------------------------------------------------*) 0000480
VAR CH : INTEGER;                                                           0000490
BEGIN                                                        (*GETLINE*)    0000500
   READLN; CH := 1;                                                         0000510
   WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN                    0000520
      READ( THISLINE(.CH.) ); CH := CH + 1;                                 0000530
      END;                              (*FILL IN REST OF LINE WITH BLANKS*) 0000540
   FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := ' ';                     0000550
END;                                                         (*GETLINE*)    0000560
PROCEDURE PRINTLABEL; FORWARD;                                              0000570
"&PRINTLABEL
PROCEDURE PRINTLABEL;
(*-------------------------------------------------------------------------- 0000580
   PROCEDURE PRINTLABEL;                                                    0000590
   PURPOSE  : PRINTLABEL PRINTS ONE COPY OF THE LABEL.                      0000600
   INPUTS   : LINESPERLABEL - NUMBER OF LINES PER LABEL                     0000610
              CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                  0000620
              LABELIMAGE - THE LABEL TO BE PRINTED                          0000630
--------------------------------------------------------------------------*) 0000640
BEGIN                                                      *PRINTLABEL*)    0000650
  FOR L := 1 TO LINESPERLABEL DO BEGIN                                      0000660
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );          0000670
     WRITELN; END;                                                          0000680
END;                                                       (*PRINTLABEL*)   0000690
BEGIN                                                    (*READANDPRINT*)  0000700
  FOR L := 1 TO LINSPERLABEL DO GETLINE( LABELIMAGE(.L.) );                 0000710
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;                                    0000720
END;                                                     (*READANDPRINT*)  0000730
BEGIN                                                          (*LABELS*)  0000740
   INTERACT; READANDPRINT;                                                  0000750
END.                                                           (*LABELS*)  0000760
```

Figure 6-11. Example of Input to SPLITPGM (Sheet 2 of 2)

6.5 SPLIT OBJECT UTILITY

The Split Object utility, SPLIT, divides the object modules in an object file into members of an object library. CONFIG automatically performs this function when the object file is input to the deferred processing run of CONFIG. SPLIT can perform the same function on an object file written by CODEGEN if the file has not been submitted to a deferred processing run of CONFIG.

The library synonym that SPLIT uses for the object library is OBJLIB. SPLIT uses the following files:

File Name	I/O	Description
OBJECT	I	Contains object file to be split
OUTPUT	O	Contains error messages
SYSMSG	O	Contains system messages

TIP software includes an SCI procedure, XSPLIT, for executing SPLIT. Enter XSPLIT when the system is ready for a command.

The following appears:

```
SPLIT OBJECT FILE INTO A DIRECTORY <VERSION: X.X.X YYDDD>

                OBJECT FILE: filename@
           OBJECT DIRECTORY: filename@
              ERROR LISTING: [pathname@]
                   MESSAGES: [pathname@] (*)
                       MODE: {FOREGROUND/BACKGROUND} (BACKGROUND)
                   COMPRESS: {YES/NO} (NO)
```

The prompts request the following information:

- OBJECT FILE — Access name of the object file as written by the compiler; if left blank, .OBJECTxx is used, where xx is the station number
- OBJECT DIRECTORY — Access name of a directory where SPLIT can place the split object modules. If OBJLIB is entered, the library associated with that synonym is used
- ERROR LISTING — Access name of the device or file for the error listing
- MESSAGES — Access name of the device or file for the system messages
- MODE — Mode of execution (foreground or background)
- COMPRESS — Option to convert the object from ASCII to compressed format. NO specifies copying the object without change. YES specifies that compressed format object will be written in the object directory. Compressed format object files occupy about 40% less space than normal ASCII format object but cannot be used on media that do not support binary data (such as cassette).

7

Link Editing and Execution

7.1 GENERAL

The topics in this section are arranged as follows:

- An introduction to link editing using simple case examples
- A discussion of linking for shared procedures
- A description of the TIP run-time options as they relate to DNOS execution
- Specific linking and execution topics for DNOS
- Compiling, linking, and executing a TIP program from a batch stream
- Overlays in TIP programs

7.2 INTRODUCTION TO LINK EDITING

The following is a brief overview of the link-editing operation. Refer to the *DNOS Link Editor Reference Manual* for details.

The Link Editor provides a means of combining separate object modules into a single linked output. A TIP program is executable when all program object modules are linked with the required run-time support modules and the linked output is installed on a program file. You can install the linked output using SCI commands or you can instruct the Link Editor to do so after linking is completed. You can also use the Link Editor to link separately installed procedure and task segments. To allow tasks to share one or two procedures, link the task and procedure segments as described in this section.

7.2.1 The Link Control File

The first step in link editing is to develop a control file that defines the link edit functions. This link control file is typically a sequential disk file created by using the Text Editor, but it can also be a card, tape, or cassette file. The link control file contains link edit commands that define the link edit operation and can also contain object modules. The basic link edit commands that are described and used in this section are as follows:

Command	Function
INCLUDE	Specifies an object module to be linked
LIBRARY	Specifies the pathname of an object library to be searched for unresolved external references from linked modules
PROCEDURE	Defines a phase that will be installed as a procedure
TASK	Defines a phase that will be installed as a task
FORMAT	Defines the format of the linked output
END	Indicates the end of the control stream

Four other commands used in this section are SEARCH, ALLOCATE, PHASE, and DUMMY, which are described as they are introduced. Consult the *DNOS Link Editor Reference Manual* for a discussion of all link edit commands.

7.2.2 Linking a Single Task Only

The most simple program structure to link is that of a single task with no procedure segments. Figure 7-1 shows an example of the link edit command list for the task only link.

```
FORMAT IMAGE
LIBRARY .TIP.OBJ
TASK <name>
INCLUDE (MAIN)
INCLUDE <user task>
END
```

Figure 7-1. Linking a Single Task Only

This is essentially the link control file built for you by the Execute TIP Compile and Link (XTIPL) command described in Section 5. Other link edit structures and the control files to produce those structures are discussed later in this section.

The FORMAT command specifies memory image output to be installed on the program file specified for the LINKED OUTPUT ACCESS NAME prompt of the Execute Link Editor (XLE) command (paragraph 7.2.3).

The LIBRARY command instructs the Link Editor to search the library with the pathname .TIP.OBJ for any unresolved external references found in the user task module. The Link Editor also searches specified libraries for modules that are enclosed in parentheses after an INCLUDE command (such as MAIN in Figure 7-1). The TIP libraries are described in paragraph 7.4.

The TASK command defines and names a phase of the Link Edit structure that will be installed as a task. The TASK command must follow PROCEDURE commands (if used) and precede the INCLUDE commands that define the task module.

The INCLUDE commands specify the modules that make up the task. The first is the MAIN module in library .TIP.OBJ. The MAIN module is a partial link of certain run-time routines that are always needed. The first module in MAIN is P$MAIN, which must always be the first item in the task segment. The second INCLUDE command specifies an object module for your task, typically the pathname of the object file created by the TIP compiler for your task.

The END command marks the end of the control stream.

7.2.3 Executing the Link Editor

After you have defined a link control file, you can begin the link edit by using the Execute Linkage Editor (XLE) command. The following prompts appear:

```
[] XLE
EXECUTE LINKAGE EDITOR
          CONTROL ACCESS NAME: pathname@ (*)
    LINKED OUTPUT ACCESS NAME: [pathname@] (*)
          LISTING ACCESS NAME: [pathname@] (*)
          PRINT WIDTH (CHARS): (80)
                   PAGE LENGTH: (59)
```

For the CONTROL ACCESS NAME, enter the pathname of the file (or device) that contains the link control file to be used.

For the LINKED OUTPUT ACCESS NAME, enter the pathname of the file (or device) to which the linked output is to be written. If you use the FORMAT IMAGE command in your link control file, the response to the LINKED OUTPUT ACCESS NAME must be a program file. If the file specified does not exist, the Link Editor creates a program file with the name specified and installs the linked output on the file. A Link Editor error and abnormal termination occurs if the file exists but is not a program file. If you enter DUMY or a blank line in response to this prompt, no output is generated. This technique allows for a trial run to check for errors.

For the LISTING ACCESS NAME, enter the pathname of the file (or device) to which the load map (also called the link map) listing is to be written. If you enter DUMY or a blank line here, no listing is generated.

The PRINT WIDTH prompt allows you either to specify the number of characters in a listing print line or to accept the default (80 characters). The PAGE LENGTH prompt allows you either to specify the maximum number of lines per page on the listing file (or device), or to accept the default (59 lines).

7.2.4 Installing Procedures and Tasks

The example link control files used in this section include the FORMAT IMAGE command, which specifies that the linked output be in memory image format. Memory image output appears exactly as the program appears in the computer memory. In addition, the FORMAT IMAGE command instructs the Link Editor to install the task or procedure on the program file specified in response to the LINKED OUTPUT ACCESS NAME prompt for the XLE command. A task is installed under the name given in the TASK command of the link control file; a procedure is installed under the name given in the PROCEDURE command of the link control file. If the task or procedure already exists on the program file under name XXXX and you want to replace the old one with the new, use the name XXXX in the TASK or PROCEDURE command within the link control file and use the following FORMAT command:

```
FORMAT IMAGE, REPLACE
```

The Link Editor will search the specified program file for an identically named task or procedure and replace it with the new linked output.

If you do not use the FORMAT IMAGE command, you can install procedures and tasks after link editing by executing the Install Procedure (IP) and Install Task (IT) SCI commands (refer to the *DNOS System Command Interpreter (SCI) Reference Manual*). The IT command allows you to specify attached procedures if appropriate. You have the option of installing attached procedures on the same program file as the associated task. The attached procedures must be installed on program file .S$SHARED if they are not on the same program file as the associated task.

7.2.5 The Execute Pascal Task (XPT) Command

Use the Execute Pascal Task (XPT) to initiate a task that uses SCI synonyms for I/O access. This is the normal method unless you use the library .TIP.LUNOBJ in the link edit as described later in this section. If the task uses LUNOs for I/O, use one of the Execute Task commands described in paragraph 7.5.4 to initiate the task.

Use XPT to execute a TIP program that uses SCI synonyms for I/O access. Enter the procedure name XPT at any time the system requests a command. The following display appears:

```
[] XPT
EXECUTE PASCAL TASK

      PROGRAM FILE: pathname@      (*)
   TASK NAME OR ID: character(s)   (*)
             INPUT: [pathname@]    (*)
            OUTPUT: [pathname@]    (*)
          MESSAGES: [pathname@]    (*)
      MODE (F,B,D): {FOREGROUND/BACKGROUND/DEBUG} (FOREGROUND)
            MEMORY: integer,integer (1,1)
```

The prompts request the following information:

- PROGRAM FILE — The access name of the program file on which the was installed
- TASK NAME OR ID — Task name supplied to the Link Editor or numeric ID associated with that name on the program file
- INPUT — The access name of the file or device for predeclared textfile INPUT
- OUTPUT — The access name of the file or device for predeclared textfile OUTPUT
- MESSAGES — The access name of the file or device for system messages
- MODE — Mode of execution (foreground, background, debug) (Entering DEBUG places the program in the suspended state. This allows you to enter any of the SCI or Pascal debug commands described in Section 9.)
- MEMORY — An ordered pair specifying stack and heap requirements for the program (in units of 1024 bytes). The default value is 1,1 initially. When a value is specified, it becomes the default value until another value is specified. The heap size parameter specifies the initial allocation; it will be expanded automatically if needed during execution. Refer to paragraph 8.2 for more information about the stack and heap

When a program accesses files other than the predeclared files INPUT and OUTPUT, the file name used in the source program is a synonym of the actual file pathname (unless procedure SETNAME or SETMEMBER has been called to specify a different synonym). You can assign a pathname as the value of the synonym by using an Assign Synonym (AS) SCI command prior to entering the XPT command. When no value has been assigned to the synonym prior to executing XPT, the run-time system assigns a default pathname. The default value is a pathname consisting of the file name with the digits of the station number appended. When the file name consists of more than six characters, the pathname consists of the first six characters of the file name and the station number. The resulting name is cataloged in the directory specified by synonym $TIP. For execution from a terminal, the default value of $TIP is a null string, indicating the system disk volume. In batch streams, $TIP is the pathname of a temporary directory. For example, if the program accesses files FILEA and DISPLAY and the program is executed from terminal ST03, the default pathnames are .FILEA03 and .DISPLA03

The run-time system displays the following message on the system message file when program execution begins:

```
<program name> EXECUTION BEGINS
```

When the program terminates normally, the run-time system displays the following:

```
NORMAL TERMINATION
```

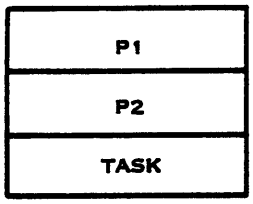
If the program terminates abnormally, the run-time system displays an error message. Following either termination message, the run-time system displays a summary of the approximate amount of stack and heap space used.

Rather than using XPT, you can write a procedure to execute a TIP task provided the procedure contains SCI primitive .BID, .QBID, or .DBID to bid the task.

7.3 LINKING FOR SHARED PROCEDURES

The DNOS operating system allows several tasks to share one or two procedure segments. This requires that the code in a shared procedure segment be reentrant and that the linking of each task that shares a procedure allows the same amount of memory for the shared procedure. The object code compiled by the TIP compiler is reentrant, and most of the modules in the run-time libraries are reentrant.

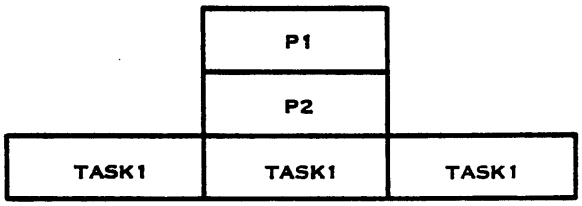
The memory mapping scheme that the operating system uses can divide programs into segments. There can be up to three segments, called procedure 1 (P1), procedure 2 (P2), and task. Figure 7-2 is a simple diagram of the segments.



2277729

Figure 7-2. Three Program Segments

Tasks can share subroutines mapped into P1 or P2, making P1 and P2 reentrant procedures (Figure 7-3).



2277730

Figure 7-3. Multiple Executions of the Same Task

In this case, TASK1 has been replicated or executed at three different terminals. Each execution requires that a new copy of the task portion reside in memory; however, since the task is sharing a reentrant P1 and P2, only one copy of P1 and P2 are memory resident. The task is said to be attached to the procedures. Thus, sharing procedures reduces the total amount of memory required in the computer to run the application.

Additional constructs are also available. Figure 7-4 shows two examples of multiple procedures being shared by multiple tasks, demonstrating two levels of complexity.

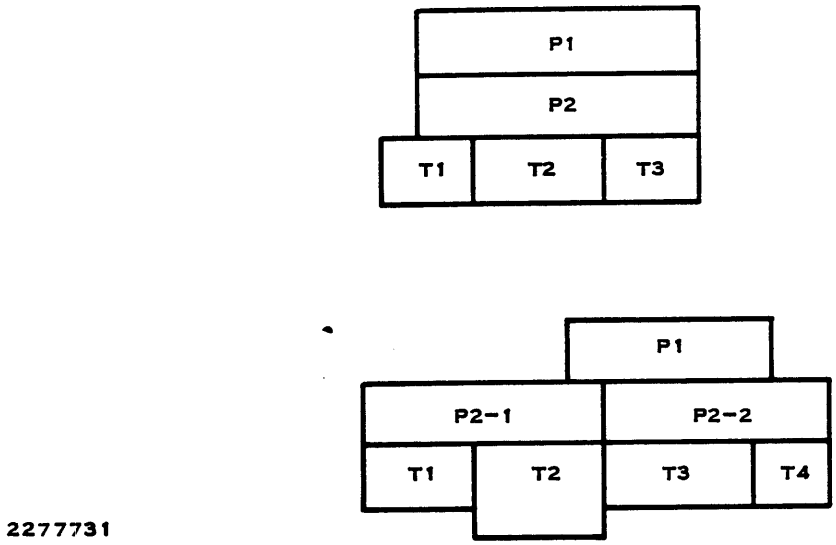


Figure 7-4. Multiple Procedures Shared by Multiple Tasks

In the lower part of Figure 7-4, both P2-1 and P2-2 share P1. Task 1 and Task 2 share P2-1 and P1, and Tasks 3 and 4 share P2-2 and P1. Sharing procedures can significantly reduce the amount of memory required to execute a group of tasks.

7.3.1 Linking a Single Procedure and a Single Task

You can use the link control file shown in Figure 7-5 to link a task with a procedure segment that is used only to share code between multiple replications of the same task. (The procedure segment is not shared with other tasks.) This represents a common linking of a TIP task.

```
FORMAT IMAGE
[LIBRARY <user library>]
[LIBRARY .TIP.LUNOBJ]
LIBRARY .TIP.OBJ
PROCEDURE <name>
INCLUDE (ENT$)
INCLUDE (GO$)
INCLUDE (MESAG$)
INCLUDE (P$TERM)
INCLUDE <user program>
SEARCH
TASK <name>
INCLUDE (P$MAIN)
END
```

Figure 7-5. Linking One Replicatable Task and a Shared Procedure

This places all of the executable TIP code in the procedure segment, with only data areas and the non-sharable module P$MAIN in the task segment. Using this structure, you minimize the memory area required for each replication of the task.

Use the SEARCH command in this case to resolve references to the library (or libraries) within the procedure phase. This sequence is necessary since automatic resolution normally occurs at the end of the entire link edit, and any modules included to resolve references would otherwise be placed after P$MAIN in the linked output. The SEARCH command explicitly controls library searching (sometimes in conjunction with the NOAUTO command); consult the *Model 990 Computer DNOS Link Editor Reference Manual.*

7.3.2 Linking a Single Procedure and Multiple Tasks

If you want different tasks to share the same P1 and if P1 contains references into the tasks, the length of references from P1 must be the same for all tasks that share it. Fixing the length of P1 references establishes the same relative address for data areas referenced by P1 for all tasks that share it.

7.3.3 The ALLOCATE Command

The Link Editor uses an ALLOCATE command to fix the data areas referenced by a procedure segment for all tasks that share it. ALLOCATE instructs the Link Editor to allocate all data areas that exist at the time the ALLOCATE command is encountered in the link stream for all modules whose routines have already been allocated.

The following rules apply to the use of the ALLOCATE command:

- When the program accesses COMMON and you link with ALLOCATE, ensure that any COMMON reference in a module that precedes the ALLOCATE is not expanded by modules that follow the ALLOCATE. The maximum size of a COMMON module defined in a shared procedure must be defined in a module included before the ALLOCATE command.

- The procedure being shared must not refer to modules or data occurring after the ALLOCATE command. For example, a subroutine in a shared procedure cannot call a subroutine included in the link edit after the ALLOCATE.

Suppose tasks TASKA and TASKB are to share procedure P1 and that P1 contains references into the task segments. The link control file shown in Figure 7-6 will correctly link TASKA and TASKB to P1.

```
FORMAT IMAGE
LIBRARY .TIP.OBJ
PROCEDURE P1
INCLUDE USER.P1
TASK TASKA
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE USER.TASKA
END
```

```
FORMAT IMAGE
LIBRARY .TIP.OBJ
PROCEDURE P1
DUMMY
INCLUDE USER.P1
TASK TASKB
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE USER.TASKB
END
```

Figure 7-6. Linking a Single Procedure, Multiple Tasks

The ALLOCATE command allocates storage to all data and common areas from the procedure when the command is encountered. Provided the fixed-length portion (P$MAIN) is included before the ALLOCATE command, tasks can share the procedure.

7.3.4 Sharing Two Procedures and Multiple Tasks

Two or more tasks can share two procedure segments (called P1 and P2) under the following conditions:

- P1 must not contain references to the other procedure or task segments.
- If P2 contains references to tasks, the length of references from P2 to the task must be fixed with the ALLOCATE command.
- A TIP task that executes under DNOS and uses LUNOs for I/O access can have two attached procedure segments that contain shared modules. When such a task includes FORTRAN routines, the FORTRAN routines must be in either the second procedure segment or the task segment.
- A task that executes under DNOS and uses SCI synonyms for I/O access can include only one attached procedure segment unless the first procedure does not contain any Pascal routines.

Linking two procedures with multiple tasks is particularly useful when an installation builds a library of procedures that all TIP users share. (This library can be added to the system program file, .S$SHARED, or maintained as a separate library.) Procedures from this library are linked in P1, since they cannot contain references to procedures or tasks outside the library. P2 can contain references to tasks provided you use the ALLOCATE command as previously described.

The link command lists shown in Figure 7-7 will properly link two tasks that share two procedure segments that meet the requirements listed above.

```
FORMAT IMAGE
[LIBRARY .TIP.LUNOBJ]
LIBRARY .TIP.OBJ
PROC P1
INCLUDE USER.P1
PROC P2
INCLUDE USER.P2
TASK TASKA
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE USER.TASKA
END
```

```
FORMAT IMAGE
[LIBRARY .TIP.LUNOBJ]
LIBRARY .TIP.OBJ
PROC P1
DUMMY
INCLUDE USER.P1
PROC P2
INCLUDE USER.P3
TASK TASKB
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE USER.TASKB
END
```

Figure 7-7. Linking Two Procedures, Multiple Tasks

7.3.5 Selecting the Shared Modules

The first step in linking shared procedures is to determine which modules can be shared. To do so, perform a linking operation for each of the tasks as if there were to be no shared procedure segment. For DNOS, the tasks are linked as described in paragraphs 7.5.1 and 7.5.3, except that they INCLUDE(P$MAIN) command replaces the INCLUDE(MAIN) command. (MAIN is a partial link that includes P$MAIN and many other run-time routines.) For minimal run time, the linking is as shown in paragraph 7.5.6, with the same substitution.

After linking each task, examine the link map listings of the tasks. The module definition portion of each map lists the modules that were linked. Compare the module lists of each listing to identify the modules that can be placed in a shared procedure segment. Use the names of these modules (or the first six characters of the names) to select the modules for the shared procedure.

The next step is to link the shared procedure. The link edit control file is as follows, with optional linking commands enclosed in brackets ([ ]):

```
[LIBRARY<user library>;]          Optional user library
[LIBRARY .TIP.MINOBJ;]            Optional run-time library
[LIBRARY .TIP.LUNOBJ;]            Optional run-time library
LIBRARY .TIP.OBJ;                 Run-time library
PROCEDURE<procedure name>;        Associates name with procedure segment
INCLUDE(<module name>);           Shared module
  .
  .
  .
SEARCH;
TASK <task name>;                 Associates name with task segment
DUMMY;                            Dummy task segment
INCLUDE (P$MAIN);
ALLOCATE;
END
```

The link control file must contain an INCLUDE command for each shared module in the procedure segment. Leave no unresolved references in the shared procedure. The linked object module that this operation produces is the linked object for the shared procedure segment. Use the IP command to install the procedure segment.

The library contains a partial link of most of the routines shared by most applications. The following command selects the linked object of this partial link:

```
INCLUDE (SHRPROC)
```

This command replaces the INCLUDE commands for the modules included in the partial link. To link the shared procedure, use the INCLUDE (SHRPROC) command; any unresolved references in the procedure segment indicate that INCLUDE commands for the modules that define those references are needed in the link control file.

The last step is to link the tasks. To do so, use the same LIBRARY commands and INCLUDE commands that linked the procedure, even though the procedure segment is a dummy segment. These commands result in the correct linking within the task segments. The link edit control file for the task contains the following commands:

```
[FORMAT IMAGE;]                          Optional format of linked object code
[LIBRARY<user library>;]                 Optional user library
[LIBRARY .TIP.MINOBJ;]                   Optional run-time library
[LIBRARY .TIP.LUNOBJ;]                   Optional run-time library
LIBRARY .TIP.OBJ;                        Run-time library
PROCEDURE<procedure name>;               Associates name with procedure segment
DUMMY;                                   Dummy procedure segment
INCLUDE(<module name>);                  Shared module
  .
  .
  .
SEARCH
TASK <task name>;                        Associates task with task segment
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE<user object>;                    TIP compiler output for main program
INCLUDE<user object>;                    Nonshared module
  .
  .
  .
END
```

The file must contain an INCLUDE command for each shared module in the procedure segment. The resulting linked object module is the linked object module for the task.

Note one exception to the rule that the LIBRARY commands used for linking tasks that share a procedure segment must be identical. To share a procedure segment with tasks that execute under DNOS when not all of the tasks access I/O in the same way (some use LUNOs, others use SCI synonyms), the following must be true:

- The shared library routines are not fundamentally different in the two libraries, .TIP.LUNOBJ and .TIP.OBJ.
- The .TIP.LUNOBJ version is used.
- Modules ENT$ and ENT$3 through ENT$16 must be those in library .TIP.LUNOBJ. To ensure this, use explicit INCLUDE commands for these modules in the link edit control file for linking the procedure segment. This is required even if only the routines in the task segment use these routines.

When you use the partial link SHRPROC, these requirements are met; that is, the modules linked in this partial link are essentially the same in both libraries. Table 7-1 lists the routines that cannot be included in the shared procedure segment. Since an open file operation calls some of these routines, the procedure segment cannot contain a user routine that opens a file.

Table 7-1. Run-Time Routines Not Sharable Between SCI and LUNO Tasks

EXTMSG	EXTND
FIND$S	EXOUT
INIT$	GO$
MESAG$	INIT$1
P$TERM	OPEN$
P$UC	REST$R
SET$NA	REST$S
TERM$	REST$T
TIP$TC	REWRT$

Note:

The modules in the left column are not sharable because they are significantly different in libraries .TIP.OBJ and .TIP.LUNOBJ. The modules in the right column are not sharable because they use routines in the left column.

To link the segments for a procedure attached to tasks using both types of I/O access, link the procedure segment using a PARTIAL command and libraries .TIP.LUNOBJ and .TIP.OBJ. No unresolved references should result except for references to labels defined in module P$MAIN. However, if the procedure segment is to be the first of two attached procedures, no unresolved references are allowed.

The link edit control files for linking the tasks should contain procedure segments that consist of only an INCLUDE command. This command specifies the module containing the partially linked object code for the procedure segment. Link each task either with or without library .TIP.LUNOBJ, as required.

Usually, a task linked with .TIP.MINOBJ cannot share run-time code with a task that does not use .TIP.MINOBJ. If a procedure segment is linked using .TIP.MINOBJ, all tasks using that segment must also use .TIP.MINOBJ.

7.3.6 Sharing Run-Time Only

Reentrant run-time code does not address a data area directly. Because of this, you can include reentrant run-time modules in P1 for a link edit that uses one or two procedure segments, as previously described. However, consider the following in regard to the MAIN module.

7.3.7 The MAIN Module

The usual TIP link control stream includes the MAIN module as the first module in the link. MAIN is a partial link of certain run-time routines that are always needed. The first module in MAIN is P$MAIN, which must always be the first item in the task segment. As a result, you do not use MAIN when linking with a procedure segment; instead, include P$MAIN in the task segment and place the rest of the run-time routines in the procedure segment, as in Figure 7-8.

```
FORMAT IMAGE
[LIBRARY .USERLIB]
LIBRARY .TIP.OBJ
PROC P1
INCLUDE (ENT$)
INCLUDE (GO$)
INCLUDE (MESAG$)
INCLUDE (P$TERM)
INCLUDE <other shared run-time routines>
[INCLUDE <user routines>]
SEARCH
TASK TASKA
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE USER. TASKA
END
```

Figure 7-8. Sharing Reentrant Run-Time Routines

7.3.8 The DUMMY Command

After the first link edit of a program file, subsequent link edits specify the reentrant run-time procedure and then use the DUMMY command to prevent the Link Editor from including a new copy of the procedure. The Link Editor requires that the procedure be specified to obtain the correct allocation. However, since the reentrant run-time procedure is already installed on the program file, the DUMMY command overrides the installation of the procedure.

7.3.9 Program Considerations for Multiple Tasks

When writing the source code for multiple tasks that communicate with each other through shared data areas, take care in handling COMMON data blocks, pointer-type variables, and global variables. The following paragraphs discuss the relevant considerations.

7.3.9.1 COMMON Data Blocks. You can link multiple tasks that execute under DNOS so that COMMON blocks are global. Alternately, you can link them so that all or part of the COMMON data is accessible to one task only (local). When you include procedure $BLOCK in a procedure segment, the COMMON blocks that it references are global to all tasks that share that procedure segment. Otherwise, COMMON blocks are linked in the task segment and are local to the task.

When you place COMMON blocks shared by several tasks in a shared procedure segment, the resulting procedure is no longer reentrant; that is, the data in the block may change. The procedure is referred to as a *dirty procedure* because it is no longer pure code. Placing a COMMON block(s) in a procedure segment requires a special $BLOCK module in the linking operation. The special module supports the BLOCK DATA statement of FORTRAN. To place one or more COMMON blocks in a procedure segment, write a TIP procedure named $BLOCK that contains an ACCESS declaration of the COMMON variables to be shared. Then, include this module in the procedure segment.

7.3.9.2 Pointer-Type Variables. Variables of pointer type should be local to one task. These variables access heap space. Under DNOS, the access to heap is by means of mapping, which differs from task to task. As a result, pointers of one task are invalid in another task.

7.3.9.3 Referencing Global Variables. When multiple tasks share procedure segments that include user routines, write the user routines with care to avoid invalid data references. A routine declared as EXTERNAL must not reference global variables. (In this context, global variables are variables declared neither in the routine nor in COMMON.) The reason for this limitation is that these global variables are accessed through the stack frame; the stack frame of a routine declared as EXTERNAL is different from the stack frame of the same routine when declared and called within the program. To avoid referencing global variables in a routine declared as EXTERNAL, use the GLOBALS option when compiling the routine.

7.4 TIP RUN-TIME OPTIONS

The run-time options relate to the capabilities that a task requires and to the operating systems under which it executes. To provide these options, three run-time libraries are linked in various ways with the object code produced by the TIP compiler. The libraries are .TIP.OBJ, .TIP.LUNOBJ, and.TIP.MINOBJ. Table 7-2 lists the capabilities and operating systems that apply to each library.

The run-time sizes shown in the descriptions of the options are minimum sizes. Incorporating additional features requires additional run-time code. Appendix C lists additional features and the amount of additional code required.

Table 7-2. TIP Run-Time Libraries

Option	.TIP.OBJ Library	.TIP.LUNOBJ Library	.TIP.MINOBJ Library
I/O access	SCI synonyms	LUNOs	LUNOs
Files INPUT and OUTPUT	Predeclared	Predeclared	Optional
Postmortem dump	Linking option	Linking option	Not available
Error diagnostics	English messages	English messages	Code numbers
Heap allocation	Allocated at start-up	Allocated at start-up	Optional; static allocation
Probe displays	Optional	Optional	Not available
Execute stand-alone	No	No	Yes
Bid using SCI	Yes	Yes	Yes
Bid by another task	No	Yes	Yes
Memory-resident task	No	No	Yes

7.4.1 Execution Under DNOS

To execute a TIP task under DNOS with full run-time support using SCI synonyms to define the I/O files and devices, link the TIP run-time library .TIP.OBJ with the object code provided by the compiler. The linking instructions are given in paragraph 7.5.1, and execution instructions are given in paragraph 7.5.2. This procedure requires linking at least 16,500 bytes of run-time code with the task. Each main program includes an overhead of 224 bytes. This overhead includes stack space for the descriptors for predefined files INPUT and OUTPUT. Thus, about 16,700 bytes are required in addition to the object code for the task. Optionally, the dump routines may be omitted, reducing the size of the run-time code to 13,000 bytes.

7.4.2 LUNO I/O

The primary advantage of defining the I/O files and devices with LUNOs instead of SCI synonyms is that the program is independent of SCI and therefore does not require run-time support in the program overhead.

Execution under DNOS with access to files and I/O devices by means of LUNOs requires linking the task with library .TIP.LUNOBJ. As a result, the overhead is somewhat less than with the .TIP.OBJ library. Specifically, the size of the minimum run-time code is approximately 15,300 bytes (11,700 bytes if the optional dump routines are not included). Programming considerations for this option are discussed in paragraph 7.5.5.

7.4.3 Multitask Capability Under DNOS

A set of task routines available on either library .TIP.OBJ or .TIP.LUNOBJ support multitask capability under DNOS. However, any task bid by another task must have been linked using library .TIP.LUNOBJ. To link for a called task, use the procedure described in paragraph 7.5.3. When the initial task is linked using library .TIP.OBJ, the task is executed as described in paragraph 7.5.2; when the initial task is linked using library .TIP.LUNOBJ, the task is executed as described in paragraph 7.5.4. Similarly, the sizes of the run-time code depend on the linking procedure followed. The size information in either of the two preceding paragraphs applies, depending on the run-time library linked.

7.4.4 Minimal Run-Time Capability

A third run-time library, .TIP.MINOBJ, allows you to include only the run-time code actually required, resulting in the minimal size of run-time code to be linked to the object code compiled by the TIP compiler. Specifically, you can use the minimal run-time library for TIP tasks that do not need the following:

- Memory dump on abnormal termination
- End-action error handling
- Linkage to FORTRAN routines that perform I/O
- Stack and heap allocation at program initiation
- Access to SCI synonyms or parameters
- Error messages in English

A further reduction is possible when the program does not declare any files and the predeclared files INPUT and OUTPUT are not required. To accomplish this, omit the main program routine and begin execution in a procedure instead, as described in paragraph 7.5.8.

The run-time code that results from using library .TIP.MINOBJ is at least 3,960 bytes in size. The reduction obtained by omitting the routines required for I/O results in a minimum of 1150 bytes of run-time code. Of this number, multiple tasks can share approximately 950 bytes. Each task requires 200 bytes that are not shared with other tasks.

7.4.5 Stand-Alone Execution

The minimal run-time library is also used for tasks that execute without operating system support (stand-alone). The Read-Only Memory (ROM) loader of the computer loads these tasks. The size of the run-time code for stand-alone programs is approximately 670 bytes. Use of arithmetic, set, run-time check, and heap management routines increases the size of run-time code. The size of the stack region must also be added to determine the total size of the run-time code.

7.5 LINKING AND EXECUTING FOR DNOS

This paragraph describes linking procedures for run-time libraries .TIP.OBJ, .TIP.LUNOBJ, and .TIP.MINOBJ and the execution of tasks linked by the procedures.

7.5.1 Linking for DNOS Execution Using SCI Synonyms

The procedure in this paragraph applies to a TIP task to be executed under DNOS. This task performs I/O operations using SCI synonyms for access to files and/or devices. The resulting task cannot be bid by another task (a cooperating task).

The following are the contents of the link edit control file for this type of linking operation:

```
[FORMAT IMAGE;]                    Optional format of linked object code
[LIBRARY<user library>;]           Optional user library
LIBRARY .TIP.OBJ;                  Run-time library
TASK <task name>;                  Associates task with name
INCLUDE (MAIN);                    Specifies required portion of run-time
                                   code
INCLUDE<user object>;              TIP compiler output
END
```

The first command is optional. When included, it specifies the format of the linked object file that the Link Editor produces. If the command is entered as shown, the Link Editor installs the task on a program file. When the FORMAT command is omitted, the linked object code is written on a sequential file in ASCII format. This code must be installed on a program file before execution.

The second command is also optional. Include it when you have a user library that contains one or more modules required for the linking operation.

The remaining commands are required. The task name operand of the TASK command becomes the task name on the linked object file. The object file operand of the INCLUDE command is the pathname of the object file that CODEGEN writes.

You must include additional commands in the file for the following cases:

- When the program calls FORTRAN routines
- When you want to include the probe data collection associated with the PROBES and/or PROBER option
- When you want to include an open-extend for file OUTPUT or SYSMSG

When the program calls FORTRAN routines, the FORTRAN library or libraries must be available for the linking operation. This requires one or more LIBRARY commands in the control file following the LIBRARY .TIP.OBJ command. The following commands access the two standard FORTRAN libraries:

```
LIBRARY .FORT78.OSLOBJ
LIBRARY .FORT78.STLOBJ
```

Other FORTRAN routines are on nonstandard libraries. Ideally, you should include the LIBRARY commands for only the library or libraries that contain the required routines. When a FORTRAN routine uses FORTRAN I/O, include the INCLUDE (FTNIO) command.

If compiler option PROBER or PROBES is turned on during the compilation, use the following commands to include the probe routines in the link:

```
INCLUDE  (PRB$INIT)
INCLUDE  (PRB$TERM)
```

You must include the following additional command when option PROBER is turned on:

```
INCLUDE  (PRB$PERF)
```

Similarly, include the following additional command when option PROBES is turned on:

```
INCLUDE  (PRB$COMP)
```

To open the OUTPUT file with EXTEND instead of with REWRITE, you must include the following additional command:

```
INCLUDE (EXTOUT)
```

To open the message file SYSMSG with EXTEND instead of with REWRITE, include the following additional command:

```
INCLUDE (EXTMSG)
```

If you do not want the message file, you can include a dummy message handler with the following command:

```
INCLUDE (NOSYSMSG)
```

Two additional options allow a slight reduction in the size of the run-time code by omitting the printing of the abnormal termination dump described in Section 9. One option saves 2,340 bytes and provides optional printing of an unformatted dump when an abnormal termination occurs. The other option saves 2,840 bytes but provides no dump when an abnormal termination occurs. For either option, substitute an INCLUDE (P$MAIN) command for the INCLUDE (MAIN) command (fifth command) in the control file. (MAIN is a partial link that includes P$MAIN and the dump routines.)

For the optional dump, add an INCLUDE command as follows:

```
INCLUDE (SPRSDUMP)
```

The optional dump is printed if byte 7 of the task segment contains a nonzero value. No dump is printed if this byte contains zero (the initial value). To alter the contents of the byte, use the Modify Memory (MM) or Modify Program Image (MPI) SCI command.

For the option that omits the dump, add an INCLUDE command as follows:

```
INCLUDE (NODUMP)
```

If the task uses SCI interface routines (S$...) besides those used by the TIP run-time, it may be necessary for a successful link to add the following command before LIBRARY .TIP.OBJ:

```
LIBRARY .SCI990.S$OBJECT
```

7.5.2 Executing Under DNOS Using SCI Synonyms

Use the XPT command (paragraph 7.2.5) to execute a TIP program in which I/O access uses SCI synonyms.

You can use a user-written procedure instead of XPT to execute a TIP task, provided the procedure contains SCI primitive .BID, .QBID, or .DBID to bid the task.

7.5.3 Linking for DNOS Execution Using LUNOs

The procedure in this paragraph applies to a TIP task that is executed under DNOS and that performs I/O operations using LUNOs for access to files and/or devices. The advantage of LUNO I/O is that the program is independent of SCI (does not use SCI synonyms to define files or devices). Therefore, the program does not require SCI in the program overhead. Either an SCI command or a cooperating task can bid the resulting task.

The following are the contents of the link edit control file for this type of linking operation:

```
[FORMAT<object format>;]          Optional format of linked object code
[LIBRARY<user library>;]          Optional user library
LIBRARY .TIP.LUNOBJ;              Run-time library
LIBRARY .TIP.OBJ;                 Run-time library
TASK <task name>;                 Associates task with name
INCLUDE(MAIN);                    Specifies required portion of run-time
                                  code
INCLUDE<user object>;             TIP compiler output
END
```

When the task calls FORTRAN routines, the FORTRAN library or libraries must be available for the linking operation. This requires the following LIBRARY commands in the control file after the LIBRARY .TIP.OBJ command:

```
LIBRARY .FORT78.DXLOBJ
LIBRARY .FORT78.OSLOBJ
LIBRARY .FORT78.STLOBJ
```

Paragraph 7.5.1 lists additional commands required when using the PROBES and/or PROBER options or when an open-extend is desired for the file OUTPUT or SYSMSG. You can add these commands to the control file. The options for the abnormal termination dump also apply.

7.5.4 Executing Under DNOS Using LUNOs

To execute tasks that use LUNOs for I/O access, use the Execute Task (XT), Execute Task and Suspend SCI (XTS), or Execute and Halt Task (XHT) SCI commands (described in the *DNOS System Command Interpreter (SCI) Reference Manual).* TIP tasks use the parameters of these commands as values that specify stack and heap requirements. (See PARM1 and PARM2 in the XT display below.)

Use XT for tasks that do not interact with the terminal and XTS for interactive tasks. Use XHT for tasks that are being debugged; the system bids these tasks and then places them in the suspended state.

To assign LUNOs prior to executing the task, use the Assign LUNO (AL) SCI command. Alternately, you can use the methods described in paragraph 7.5.5 to assign LUNOs within the task.

After you enter the XT, XTS, or XHT command, the following display appears:

```
PROGRAM FILE OR LUNO:  filename@ or integer
     TASK NAME OR ID:  character(s)
              PARM1:   integer          (0)
              PARM2:   integer          (0)
         STATION ID:   stationname      (ME)
```

Respond to PROGRAM FILE OR LUNO by entering either the pathname of the program file on which the task was installed or the LUNO assigned to the program file. If the FORMAT IMAGE command was placed in the link control file when the task was linked, use the pathname entered in response to LINKED OUTPUT ACCESS NAME in the XLE command.

Respond to TASK NAME OR ID by entering either the task name or the installed ID of the task. The TASK (or PHASE 0) command of the link edit control file specifies the task name; the installed ID is listed on the link edit map when the link edit operation installs the task.

PARM1 and PARM2 are integers that specify the number of bytes of memory. PARM1 requests memory for stack space, and PARM2 requests memory for heap space. In each case, the item is interpreted as a number of bytes, not thousands of bytes (as in the XPT procedure). When the default value (0) is used, 1,024 bytes are requested for stack and 1,024 bytes are requested for heap.

The prompt STATION ID requests the number of the station (terminal) with which the task is to be associated.

You can write an SCI procedure to assign the LUNOs and execute the task. (Refer to the *DNOS System Command Interpreter (SCI) Reference Manual*). Within the SCI procedure, the task must be bid by an XT, XTS, or XHT command, not by the SCI .BID, .QBID, or .DBID primitives.

7.5.5 Program Considerations for LUNO I/O Under DNOS

Access to I/O devices and/or files by a LUNO is available optionally for tasks executing under DNOS. This method of access requires that a LUNO be defined for each I/O operation, either explicitly or by default.

The SETLUNO procedure (see Section 10) is the means of explicitly defining a LUNO for an I/O operation. When no LUNO is explicitly defined, the ASCII code for the first letter of the file name is the LUNO, by default. This results in LUNOs in the range of >41 through >5A, and >24 (for file names that start with $).

The LUNOs for the predeclared files are as follows:

- >3C for SYSMSG
- >3D for INPUT
- >3E for OUTPUT

A library module defines the LUNOs for predeclared files. You can assemble a module to define other LUNOs for these files. The following is the source code for module LU$MSG:

```
          IDT   'LU$MSG'
          DEF   LU$MSG,LU$IN,LU$OUT
LU$MSG    EQU   >3C              MESSAGE FILE
LU$IN     EQU   >3D              "INPUT"
LU$OUT    EQU   >3E              "OUTPUT"
          END
```

NOTE

LU$MSG cannot contain a DEF for LU$IN if the module SHRPROC is also included in the link edit, since a double definition occurs.

To change the LUNOs for these files, change the operands for the three EQU directives to the desired LUNOs. Then, assemble the source code and link the new module with an INCLUDE command in the link edit control file.

Another way to change the LUNO for predeclared file OUTPUT is to use procedure SETLUNO along with the CLOSE and REWRITE procedures. This must be done in all tasks that attempt to use the LUNO concurrently. For example, if more than one task terminates abnormally and attempts to write abnormal termination dumps at the same time, each must have a different LUNO defined for file OUTPUT. Use the following procedure calls at the beginning of each task:

```
CLOSE (OUTPUT);          (*Close OUTPUT file (assigned to DUMY)*)
SETLUNO (OUTPUT, XX);    (*XX is desired LUNO*)
REWRITE (OUTPUT);
```

When the tasks are executed, LUNO >3E must be assigned to DUMY. Each task closes DUMY (which the run-time code automatically opens), redefines the LUNO for the file OUTPUT, and opens the file or device assigned to that LUNO.

A program can call both SETLUNO and SETNAME procedures. When the run-time code for SCI I/O is linked with the task, procedure SETNAME executes and procedure SETLUNO is ignored. When the run-time code for LUNO I/O is linked with the task, procedure SETLUNO is executed and procedure SETNAME is ignored.

You can also use procedure SET$ACNM to specify the pathname (access name) for a file. Call procedure SET$ACNM before calling the RESET, REWRITE, or EXTEND procedure that opens the file. To use SET$ACNM, declare type SCI_STRING and declare the procedure externally, as follows:

```
CONST SL=80;
TYPE SCI_STRING = PACKED ARRAY[1..SL] OF CHAR;
PROCEDURE SET$ACNM (VAR F:<ft>; VAR ACNM: SCI_STRING); EXTERNAL;
```

The <ft> (file type) entry in the PROCEDURE declaration can be TEXT or a user-defined file type. The first parameter is a file name and the second is a string that contains the access name. The first element of the array is a binary number representing the length of the string. (See Section 10 for examples.) The access name parameter cannot be a synonym. SET$ACNM associates the specified access name with the specified file.

When a routine that uses LUNO I/O calls procedure SET$ACNM, SET$ACNM associates the access name with the specified file and causes the RESET, REWRITE or EXTEND operation that opens the file to assign the LUNO to the access name. Unless you have defined a LUNO for the file by using the SETLUNO procedure, the default LUNO is used. When the RESET, REWRITE, or EXTEND operation assigns the LUNO, the CLOSE operation releases the LUNO.

7.5.6 Linking Minimal Run Time Under DNOS

Programs that meet the restrictions listed in paragraph 7.4.6 and 7.5.5 can be linked with the minimal run-time library .TIP.MINOBJ, which results in a smaller amount of run-time code due to the omission of debugging support. It also makes possible the omission of the TIP I/O support routines, as described in paragraph 7.5.8. In addition to using the minimal run-time library, you can minimize the program memory requirements by turning off the compiler options for debugging. Include the following option comment:

```
(*$ NO TRACEBACK, NO ASSERTS, GLOBALOPT*)
```

You probably do not want to include any of the following options:

CKINDEX
CKOVER
CKPREC
CKPTR
CKSET
CKSUB
CKTAG
PROBER
PROBES

The size of the required stack and heap space is specified in an assembly language module that must be assembled and linked with the object code from the compiler. A default module is provided that specifies 1024 bytes each for stack and heap space. When the default size is not appropriate, prepare an assembly language source module as follows:

```
          IDT     '<name>'              Module name
STACK     EQU     <stack size>          Number of bytes for stack
HEAP      EQU     <heap size>           Number of bytes for heap
          DEF     ST$BOT,ST$TOP,HP$BOT,HP$TOP
          DSEG
          EVEN
ST$BOT    BSS     STACK                 Bottom of stack region
          BSS     64                    Stack margin area
ST$TOP    EVEN                          Top of stack region
HP$BOT    BSS     HEAP                  Bottom of heap region
HP$TOP    EVEN                          Top of heap region
          DEND
          END
```

Obtain the stack and heap sizes for the EQU statements by executing the program under DNOS with full run time, noting the sizes returned in the termination message. You can also determine the sizes by adding the stack frame sizes of individual routines and then adding two bytes to the total for overhead. The heap size is zero if routine NEW is not called in the task and the TIP I/O routines are not used. When using the default heap manager, add the sizes of heap requested. When using the heap manager with space recovery, add the sizes of memory packets requested when NEW routines are called and not released by calls to the DISPOSE routine when the highest number of packets are active. If only one packet is active at a time, use the size of the largest packet. Add an overhead of six bytes plus two bytes per active packet.

Assemble the module, storing the module on a file to be accessed in the link edit control file during the linking operation. The link edit control file contains the following commands:

```
[FORMAT<object format>;]          Optional format of linked object code
[LIBRARY<user library>;]          Optional user library
LIBRARY .TIP.MINOBJ;              Run-time library
LIBRARY .TIP.LUNOBJ;              Run-time library
LIBRARY .TIP.OBJ;                 Run-time library
TASK <task name>;                 Associates name with task
INCLUDE (MAIN)
[INCLUDE<file name>;]             Required for object module specifying
                                  stack and heap size
INCLUDE<user object>;             TIP compiler output
END
```

NOTE

If the run time is in a procedure segment, also include the module specifying stack and heap sizes in the procedure segment.

Another optional command provides a heap manager routine that recovers heap space when the DISPOSE routine is called. The command is INCLUDE(DISPOSE). It is placed in the file following the INCLUDE(MAIN) command. When the optional command is omitted, the default heap manager is included. The default heap manager includes a limited DISPOSE routine that disposes of only the most recently allocated packet. A call to DISPOSE that specifies any other packet does not release the packet's heap space. When the optional command is included in the link edit control file, a larger heap manager is included. This heap manager recovers heap space by releasing the space allocated to any packet specified in a DISPOSE call.

7.5.7 Executing Minimal Run-Time Under DNOS

Executing TIP tasks that have the minimal run-time code linked with them requires the same SCI commands as those used to execute tasks that access I/O devices and files with LUNOs. The parameter operands of the XT, XTS, or XHT SCI commands are not used to specify stack and heap space, which are specified during the linking operation (other than this, operation is the same as descibed in paragraph 7.5.4). When the tasks are compiled with a dummy main routine and execution begins in a procedure, parameters placed in the appropriate Execute Task command or Bid Task supervisor call (SVC) are passed to the initial procedure. The declaration of that procedure must specify the desired parameters if access to them is required. Note that these parameters do not correspond to the parameters of the .BID SCI primitive, which should not be used with programs using minimal run-time code.

For tasks linked with minimal run-time which perform I/O, any required LUNOs must be assigned prior to execution (refer to paragraphs 7.5.4 and 7.5.5).

7.5.8 The Dummy Main Routine

Normally, a TIP task consists of a main routine, or program, and one or more routines that the main routine calls, directly or indirectly. A TIP task linked with the MINOBJ library can be compiled with a dummy main routine and begin execution in a procedure in either of the following cases:

- When a task linked for minimal run-time code does not declare any files and does not use the predeclared files INPUT and OUTPUT
- When a task executes as stand-alone

When a task does not use TIP I/O, you can reduce the run-time size by omitting the I/O support routines. When a task is linked with a dummy main routine, the automatic references for files INPUT and OUTPUT are omitted.

To compile a program with a dummy main routine, declare no variables in the declaration section of the main program and place a (*$NO OBJECT*) option comment in the statements section of the main program. The following are the statements of a task with a dummy main routine:

```
PROGRAM <name>;          (*Arbitrary name; not used *)
CONST ...                (*Global constant declarations*)
TYPE ...                 (*Global type declarations*)
COMMON ...               (*Declare all COMMON blocks here*)
PROCEDURE <name>;        (*Declaration of initial procedure — see text*)
    VAR ...
    PROCEDURE ...        *Declaration of additional procedure*)
    .
    .
    .
    BEGIN
    .
    .
    .
    END;
    .
    .
    .
BEGIN                    (*Statements of main program*)
  (*$ NO OBJECT*)        (*The main program is a dummy*)
END.                     (*End of program*)
```

The declaration of the initial procedure for a task that executes as stand-alone or under DNOS should use the name PSCL$$; this name identifies the procedure as the one in which execution begins.

7.6 LINKING FOR STAND-ALONE EXECUTION

A task linked for minimal run-time code can be executed stand-alone. A stand-alone task must consist of a dummy main routine as described in paragraph 7.5.8. No variables can be declared for the main routine, and the statement section includes a NO OBJECT option comment. Execution begins in a procedure named PSCL$$ (also described in paragraph 7.5.8).

Because there is no operating system, none of the TIP I/O facilities are available. To perform I/O with CRU devices, use the direct CRU I/O routines described in Section 10. To access TILINE addresses, use a type transfer to assign an integer constant to a pointer variable, used as the TILINE address. (TILINE is a trademark of Texas Instruments.) The following is an example:

```
TYPE TREC = RECORD ... END;          (*TILINE control block*)
VAR TP: @TREC                        (*TILINE address*)
  .
  .
  .
TP::INTEGER := #F800;                (*Set TILINE address*)
WITH TP@ DO BEGIN
  .
  .
  .
            END
```

With the record defined according to the requirements of the TILINE device controller and the pointer set to the TILINE address, the TILINE operations consist of assigning values to the elements of the record and of performing other appropriate operations as if the record were in memory.

The task must initialize any required interrupt or extended operation (XOP) transfer vectors. The technique that you use to assign a value to a pointer to be used as a TILINE address may also be used to assign an absolute memory address at which to store a transfer vector. The TYPE declaration declares a record that represents a transfer vector rather than a TILINE control block. The type transfer assignment sets the absolute address of the transfer vector.

Do not call any of the following library routines in a stand-alone task:

ACTIVATE	READ
BID	RESET
DATE	REWRITE
DELAY	RLSCOM
GETMSG	STATIONID
IDATE	SUSPEND
ITIME	SVC$
MESSAGE	SYSCOM
OVLY$	TASKID
PRGMSG	TIME
PUTMSG	WRITE
	WRITELN

When a stand-alone task calls SVC$, the second byte of the SVC block is set to >FF. When a stand-alone task calls MESSAGE, the label T$MSG is listed as an unresolved reference in the link map written for the linking operation. A call to any of the other library routines produces unpredictable results.

A library routine allows a stand-alone task to display a value on the programmer panel indicators. To use the routine, the task must declare the routine externally, as follows:

```
PROCEDURE DSPLY$ (VALUE: INTEGER); EXTERNAL;
```

The parameter is a value displayed by the indicators on the programmer panel as a 16-bit binary number.

7.6.1 Link Control File for Stand-Alone Tasks

The link edit control file for the linking operation is as follows:

```
FORMAT ASCII;                          Loader requires ASCII format
LIBRARY .TIP.MINOBJ;                   Run-time library
LIBRARY .TIP.LUNOBJ;                   Run-time library
LIBRARY .TIP.OBJ;                      Run-time library
PHASE 0,<name>;                        Associates name with task
INCLUDE(P$MAINSA);                     Specifies portion of run time
                                       required for stand-alone
INCLUDE<user program>;                 TIP compiler output
END
```

Stand-alone tasks can reside in a combination of ROM and Random-Access Memory (RAM), which requires the use of the PROGRAM, DATA, and COMMON Link Editor commands described in the *Link Editor Reference Manual*.

7.6.2 Stand-Alone Execution

Execution procedures for stand-alone tasks vary considerably. Typically, the ROM loader loads the linked object module. The module begins execution when the loader passes control to the entry point after the loading operation completes.

A stand-alone task terminates with an IDLE instruction followed by a JMP$ instruction (a single instruction loop). When the task has executed and the computer is executing either of these instructions, workspace register 0 contains the termination code and workspace register 1 contains the address of the process record. The termination code is one of the codes listed and defined in Appendix B; it is zero for normal termination. Section 8 lists the contents of the process record. The contents of the workspace pointer (WP) at relative bytes >22 and >23 of the process record are the address of the stack frame of the routine that executed last. The contents of the program counter (PC), bytes >24 and >25, are the address following the last instruction executed.

7.7 COMPILING, LINKING, AND EXECUTING WITH A BATCH STREAM

You can execute the TIP compiler, Link Editor, and resulting TIP program by using a batch stream. Figure 7-9 shows the SCI commands for a batch stream to compile, link, and execute a program. The example implies the existence of directory .USER, which contains the source code in file .USER.SOURCE. The procedure creates files .USER.OBJECT, .USER.LISTING, .USER.MESSAGE, .USER.LINK, .USER.PROGRAM, .USER.LINKLIST, and .USER.OUTPUT. The input data for the program must be on file .USER.INPUT.

Synonym $$CC stores system and user condition codes. Test or store the synonym prior to executing any of the system utilities that set the synonym. Figure 7-9 shows an .IF SCI primitive testing the value of $$CC within the batch stream after execution of the compiler and again after execution of the Link Editor. To store the value of $$CC after a given task and then continue batch execution, set another synonym to the value of $$CC and test later. For example, use the .SYN primitive to set synonym S to the value of $$CC, as follows:

```
.SYN S = @$$CC
```

```
  BATCH      ! Compile, link, and execute a TIP program
*
*     Execute the TIP compiler
*
  XTIP    SOURCE=.USER.SOURCE, OBJECT= .USER.OBJECT,
          LISTING=.USER.LISTING, MESSAGES=.USER.MESSAGE
*
  .IF @$$CC, GT, 04000                          ! If there are compile errors,
     EBATCH                                     !   If then stop.
  .ENDIF
*
*     Generate a link control file as input to the Link Editor
*
  .DATA  .USER.LINK                             ; Install directly into user program
  FORMAT IMAGE, REPLACE                         ; file, replacing existing task by
  LIBRARY  .TIP.OBJ                             ; the same name, or install at first
  PHASE 0,SAMPLE                                ; available location if task does
  INCLUDE (MAIN)                                ; not already exist.
  INCLUDE .USER.OBJECT
  END
  .EOD
*
*     Execute the 990 Linkage Editor
*
  XLE     CONTROL ACNM = .USER.LINK,
          LINKED OUTPUT = .USER.PROGRAM,
          LISTING ACNM = .USER.LINKLIST
*
  .IF @$$CC, GT, 04000                          ! If errors in link,
     EBATCH                                     ! then stop.
  .ENDIF
*
*  Execute the TIP program just installed.
*
  XPT     PGM FILE = .USER.PROGRAM, TASK NAME = SAMPLE,
          INPUT =  .USER.INPUT,  OUTPUT = .USER.OUTPUT,
          MESSAGE =  .USER.MESSAGE

*
  EBATCH      ! End of batch stream.
```

Figure 7-9. Batch Stream to Compile, Link, and Execute a TIP Program

7.8 OVERLAYS IN TIP PROGRAMS

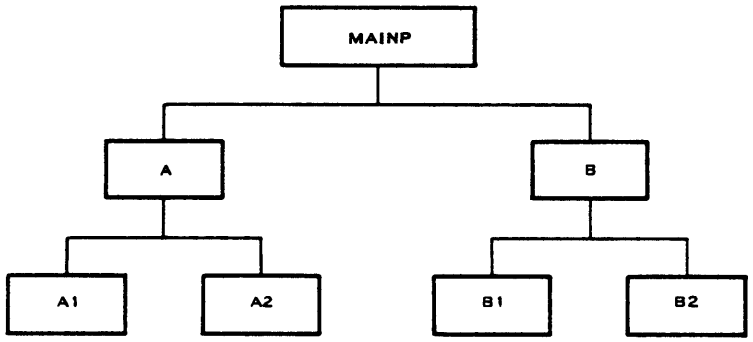
The paragraphs that follow discuss overlay structure, procedure OVLY$, and the linking and execution considerations related to overlays.

7.8.1 General

TIP software allows defining one or more modules of a TIP object program as an overlay or overlays and loading overlays appropriately during the execution of the program. Specifically, the software includes a procedure for loading an overlay. CONFIG or SPLIT must place the object code in a library of object modules. The modules must be linked in the desired overlay structure. This paragraph describes how to organize a program into overlays, the use of the overlay procedure OVLY$, the control file for linking the modules into the desired structure, and the SCI commands that install the program.

7.8.2 Overlay Structure

To illustrate the overlay structure that a TIP program could have, the following tree structure shows a program MAINP with procedures A, A1, A2, B, B1, and B2:



Program MAINP calls procedure A or procedure B; procedure A calls procedure A1 or procedure A2; and procedure B calls procedure B1 or procedure B2. The declarations for the TIP program corresponding to this structure are as follows:

```
PROGRAM MAINP;
VAR PATH:INTEGER;
PROCEDURE A;
   PROCEDURE A1;
   BEGIN              (*A1*)
     WRITELN('PATH TAKEN = 1');
   END;               (*A1*)

   PROCEDURE A2:
   BEGIN              (*A2*)
     WRITELN('PATH TAKEN = 2');
   END;               (*A2*)

BEGIN                 (*A *)
  CASE PATH OF        (*KIND OF PATH*)
     1: A1;
     2: A2;
     END;
END;                  (*A *)
PROCEDURE B;
   PROCEDURE B1;
   BEGIN              (*B1*)
     WRITELN('PATH TAKEN = 3');
   END;               (*B1*)
PROCEDURE B2;
BEGIN                 (*B2*)
     WRITELN('PATH TAKEN = 4');
   END;               (*B2*)
BEGIN                 (*B *)
  CASE PATH OF        (*KIND OF PATH*)
     3: B1;
     4: B2;
     END;
END;                  (*B *)
BEGIN                 (*MAINP*)
  READLN(PATH);       (*WHICH PATH*)
  CASE PATH OF
     1,2: A;
     3,4: B;
     END;
END.                  (*MAINP*)
```

Use overlays to implement this structure, with program MAINP as the root, modules A and B as overlays, modules A1 and A2 as overlays called in module A, and modules B1 and B2 as overlays called in module B. The modules are assigned the following overlay numbers:

Module	Overlay Number
A	1
B	2
A1	3
A2	4
B1	5
B2	6

This example illustrates a program that has an overlay structure identical to the calling structure. However, the TIP software does not require that the overlay structure coincide with the calling structure. For example, procedures B1 and B2 need not be nested within procedure B; instead, they can be declared following the declaration of procedure A.

7.8.3 Procedure OVLY$

The overlay handler provided in the TIP software is procedure OVLY$. OVLY$ is called in the module that calls the overlay prior to the call of the procedure in the overlay. Declare the procedure in the main program, as follows:

```
PROCEDURE OVLY$(LOAD:INTEGER); EXTERNAL;
```

Use the following call to initialize the handler:

```
OVLY$(0);
```

This call must precede any call to OVLY$ to load an overlay. The argument 0 causes the procedure to initialize the handler. The argument is the overlay number in subsequent calls to OVLY$ (to load an overlay). The code for the example program, including appropriate calls to OVLY$, is as follows:

```
PROGRAM MAINP;
VAR PATH:INTEGER;
PROCEDURE OVLY$(LOAD:INTEGER);EXTERNAL;
PROCEDURE A;
   PROCEDURE A1;
   BEGIN              (*A1*)
     WRITELN('PATH TAKEN = 1');
   END;               (*A1*)
   PROCEDURE A2;
   BEGIN              (*A2*)
     WRITELN('PATH TAKEN = 2');
   END;               (*A2*)
BEGIN                 (*A *)
  CASE PATH OF        (*KIND OF PATH*)
    1:BEGIN
        OVLY$(3);A1;END;
    2:BEGIN
        OVLY$(4);A2;END;

END;                  (*A *)

PROCEDURE B;
   PROCEDURE B1;
   BEGINB1*)
     WRITELN('PATH TAKEN = 3');
   END;               (*B1*)
   PROCEDURE B2:
   BEGIN              (*B2*)
     WRITELN('PATH TAKEN = 4');
   END;               (*B2*)
BEGIN                 (*B *)
  CASE PATH OF        (*KIND OF PATH*)
    3:BEGIN
        OVLY$(5);B1;END;
    4:BEGIN
        OVLY$(6);B2;END;

END;                  (*B *)
BEGIN                 (*MAIN*)
  OVLY$(0);           (*INITIALIZE HANDLER*)
  READLN(PATH);       (*WHICH PATH*)
  CASE PATH OF
    1,2:BEGIN
        OVLY$(1);A;END;
    3,4:BEGIN
        OVLY$(2);B;END;
    END;
END.                  (*MAINP*)
```

Notice that the first statement of the main program calls OVLY$ to initialize the handler. When PATH is one, overlay 1 is loaded by a call to OVLY$ with a parameter of 1, and procedure A is called. In procedure A, the call to OVLY$ loads overlay 3 and is followed by a call to procedure A1.

Procedure OVLY$ maintains a variable that contains the number of the most recently loaded overlay and compares the argument in the call with that variable. A subsequent call to load the most recently loaded overlay is ignored.

7.8.4 Link Control File for Overlays

The object modules must be properly linked to support the overlay structure. The object modules must be available as members of a library, which may be written by utility CONFIG or SPLIT. Assuming that the modules have been cataloged as members of library .USER.OBJECT, the following example shows the commands to link the program in the desired overlay structure:

```
LIBRARY .USER.OBJECT
LIBRARY .TIP.OBJ
PHASE 0,OVLYTEST
INCLUDE (MAIN)
INCLUDE (MAINP)
PHASE 1,OVERLAY1
INCLUDE (A)
PHASE 2,OVERLAY3
INCLUDE (A1)
PHASE 2,OVERLAY4
INCLUDE (A2)
PHASE 1,OVERLAY2
INCLUDE (B)
PHASE 2,OVERLAY5
INCLUDE (B1)
PHASE 2,OVERLAY6
INCLUDE (B2)
END
```

These commands are supplied to the Link Editor in the link control file, and the Link Editor writes the linked object to a file specified for the linked output. Do not use the Link Editor LOAD command, since it does not apply to TIP programs; use procedure OVLY$ instead.

7.8.5 Installing a Program With Overlays

The root module and the overlay modules must be installed in a program file. The following is an example of the batch stream of SCI commands to install the program. Assume that the linked object is on file .USER.LOBJ and that the program file on which the modules are written is .USER.OTEST. The Install Overlay commands must be in the same order as the PHASE commands in the link. Notice that the Install Overlay (IO) command assigns the overlay numbers used in the calls to OVLY$. The batch stream is as follows:

```
BATCH
.SYN P=".USER.OTEST"
AL AN=.USER.LOBJ
DT PF=P,TN=OVLYTEST
IT PF=P,TN=OVLYTEST,OBJ=@$$LU
IO PF=P,ON=OVERLAY1,OI=1,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
IO PF=P,ON=OVERLAY3,OI=3,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
IO PF=P,ON=OVERLAY4,OI=4,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
IO PF=P,ON=OVERLAY2,OI=2,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
IO PF=P,ON=OVERLAY5,OI=5,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
IO PF=P,ON=OVERLAY6,OI=6,OBJ=@$$LU,REL=NO,ASS=OLVYTEST
RL L=@$$LU
EBATCH
```

8

Internal Structures

8.1 GENERAL

This section describes the data storage areas known as stack and heap and discusses the stack frame, the heap, the process record, the file descriptor, and the supervisor call (SVC) block. This information is useful for reading memory dumps and debugging TIP programs.

8.2 STACK AND HEAP DESCRIPTION

Stack and heap are areas of memory allocated for variables within a TIP program. All variables declared in a program (static variables), except for COMMON variables, are allocated memory from stack memory. Within the stack memory, storage locations for a particular routine are not allocated until the routine is called. When a routine exits, storage for that routine is released.

Storage locations for dynamic variables, those allocated by the procedure call NEW, are in heap memory. The heap is a memory pool that can be used by a program to allocate temporary memory. The heap is also used for file buffers and control blocks.

8.2.1 Scope and Extent

The scope and extent of identifiers is related to the allocation of space for the data in routines. Within the stack, the space used for the parameters, local variables, and temporary values of a single routine is called the *stack frame*. When execution of the routine is completed, the stack frame is deallocated and its space is available to other routines.

The method of space allocation makes the identifiers accessible for a given period of time, called an *extent*. The extent of statically defined quantities is the duration of the execution of the unit of scope in which they are declared. The extent of dynamically allocated variables is the duration of program execution between the call of NEW, which creates the variable, and the call of DISPOSE, which frees the heap space allocated to the variable. When DISPOSE is not called, the extent continues to program termination. (When the minimal run-time is used, space recovery on DISPOSE calls is a link edit option.)

8.2.2 Estimating Stack and Heap Requirements

A straightforward way to estimate the stack and heap requirements of a program is to execute the program first, using the default stack and heap allocations. The default values for the MEMORY field of the XPT command are 1 and 1 (thousand bytes), initially, for stack and heap. (When a value is entered, it becomes the default value until another value is entered.) The program termination messages include the actual size of the stack and heap used (in bytes). If the program does not execute with the default memory allocation, increment the memory allocations until execution is successful. The stack and heap values returned at the end of the successful execution become the memory allocation requested for subsequent executions.

8.3 DATA STRUCTURES

The paragraphs that follow describe the stack frame, the process record, the TIP file descriptor, and the SVC block. In describing these data structures, WP stands for workspace pointer, PC stands for program counter, and ST stands for the status register.

8.3.1 The Stack Frame

The space used for the parameters, local variables, and temporary values of a single routine is called a *stack frame*. The variables for the main program are organized in a stack frame, as are those of each routine. When execution begins, the stack frame of the main program is placed in the stack. When a routine is called, its stack frame is added to the stack. If the routine calls a nested routine, the stack frame of the nested routine is also added. At any time, the stack contains the stack frame of the currently executing routine and those of all of its ancestors.

The stack frame is organized as shown in Figure 8-1.

	0	2	4	6	8	A	C	E
00	R0	R1	R2	R3	R4	R5	R6	R7
10	R8	BOTTOM	TOP	R11	R12	R13	R14	R15
20	SAVED	CALLER	RETADR	PRP	RESULT / ARGS / LOCALS / TEMPS			

2277733

Figure 8-1. The Stack Frame

The contents of the stack frame are as follows:

- R0–R15 — Workspace registers.
- BOTTOM — Bottom (beginning) of stack frame (dedicated register).
- TOP — Top-of-stack (dedicated register).
- R12 — Display pointer (contains address of display on entry to a procedure).
- R13 — Global values (standard linkage); saved WP (short or medium linkage).
- R14 — Global values (standard linkage); saved PC (short or medium linkage).
- R15 — Global values (standard or medium linkage); saved ST (short linkage).
- SAVED — Saved display pointer (standard linkage). The routine being entered saves the old display pointer at its level in this field prior to storing its bottom-of-stack pointer in the display. Used only by routines containing nested routines.
- CALLER — Caller's WP (standard).

- RETADR — Caller's PC (standard); #0000 (short); #FFFF (medium).
- PRP — Process record pointer; used with LUNOBJ library only.

Variable Length Area

- RESULT — Function result, if any (eight or ten bytes, left-justified).
- ARGS — Arguments.
- LOCALS — Local variables.
- TEMPS — Compiler-generated temporaries.

Section 11 includes a discussion of the stack frame contents as related to the use of assembly language routines.

8.3.2 The Heap Structure

The heap contains the storage areas for dynamic variables. (The procedure call NEW allocates dynamic variables during program execution, and the procedure call DISPOSE deallocates them.) The heap also contains file buffers and control blocks. The following general description will acquaint you with the aspects of the heap that are most useful in debugging.

A *heap region* is a block of memory obtained for the heap as a common area or obtained through a request to the operating system. Within a heap region, a *heap packet* is a block of storage to which a program has (or can gain) access by using the procedure call NEW. Several packets linked together form a *packet list.*

The term *statically allocated* refers to structures that are effectively created when a task is loaded. The term *dynamically allocated* refers to structures whose location and size are determined during execution. The *heap control bock* and the chains of pointers that begin there define the heap structure.

8.3.2.1. The Heap Control Block. The heap control block is referenced either as the COMMON block HEAP$ or by the pointer to it (PHEAP) in the process record. HEAP$ is defined as follows:

```
HEAP$  :  RECORD
     ROVER     : ADDRESS;  pointer to the list of heap packets
     REGION    : ADDRESS;  pointer to the list of heap regions
     HEAPLIST  : ADDRESS;  pointer to the static heap region
     CURUSED   : INTEGER;  number of currently used bytes of heap
     MAXUSED   : INTEGER;  maximum value reached by CURUSED
     MUTEX     : INTEGER;  not used
     USEFLAG   : BOOLEAN;  switch; controls heap use statistics
     END;
```

The following describes the relevant fields of HEAP$:

- ROVER — Provides access to the heap packet list.
- REGION — Initially points to the header block of the static region. It is used to keep track of dynamic allocations and deallocations.

- HEAPLIST — Provides access to the heap region list (see paragraph 8.3.2.2).
- CURUSED — Contains the number of bytes defined as allocated at any time. This number does not include the packets allocated for the process record and the stack regions of the program process.
- MAXUSED — Contains the largest value attained by CURUSED. The termination routine displays this value as the amount of heap used.

8.3.2.2. The Heap Region. The packets that make up the heap are contained in heap regions. The heap initially includes one statically allocated region. As requests for packets exhaust the space available, at least one dynamically allocated region is added. (However, with the library .TIP.MINOBJ there is only one statically allocated region.) The COMMON HEAP$ contains field HEAPLIST, which points to the header block of the static region. The header block for a region contains the following fields:

- HEAPMIN — Points to the low end of the region.
- HEAPMAX — Points to the high end of the region.
- NEXTREGION — Points to the header block of the next region.

The TIP run-time support package uses HEAPMIN and HEAPMAX to check packet pointers and length fields for obvious invalidity. The NEXTREGION pointers form a circularly linked list of all heap regions. Following the header block is the region from which packets are allocated. Figure 8-2 shows the heap region structure with one dynamic region. The figure shows the pointers HEAPMIN, HEAPMAX, and NEXTREGION.

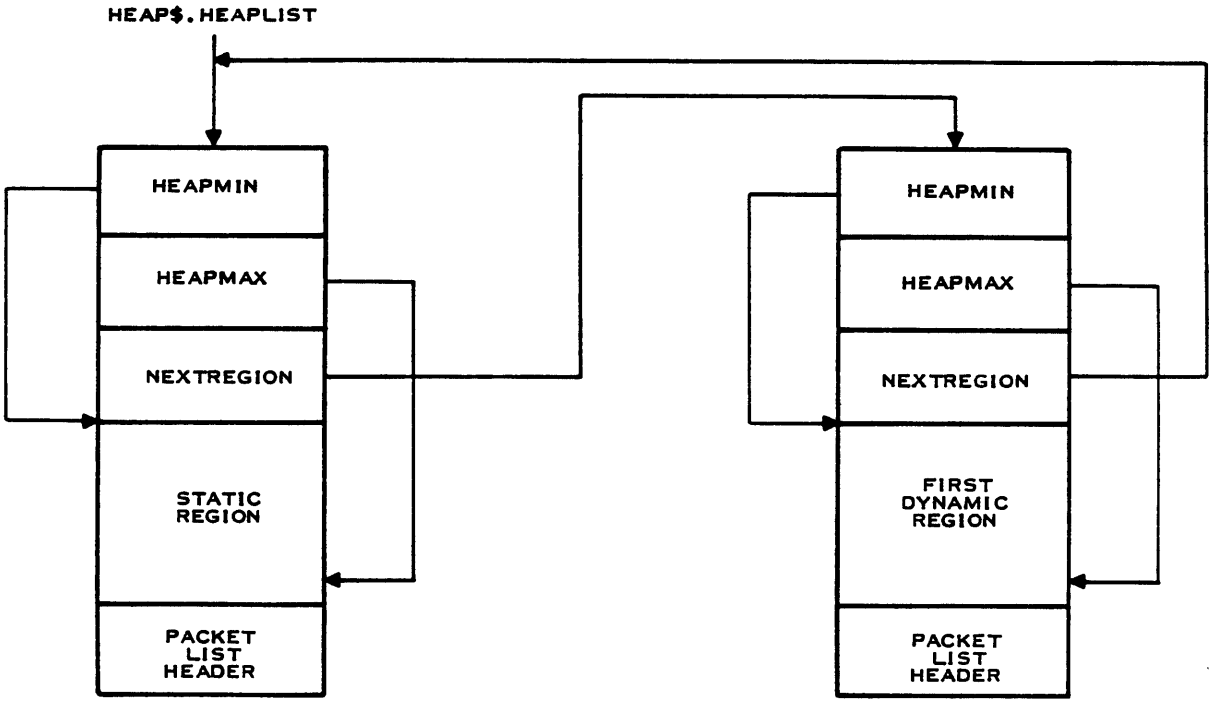


Figure 8-2. The Heap Structure

A length word containing an odd number precedes each allocated packet. This number is the size of the packet plus 1. (The size is in bytes and includes the length word.) Each unallocated packet has a length word containing an even value. These packets are chained together in a doubly linked list. Two words (called predecessor and successor) follow the length word and point to the previous and succeeding packets in the packet list.

8.3.3 The Process Record

The TIP run-time support system permits multiple sites of execution within the executable object code of a program. Each of these sites is called a *process* and has its own stack region within which its local variables are allocated. A process dump consists of the process record and the stack. The process record is the structure that describes the resources associated with each process. The fields of the process record are shown in Figure 8-3.

	0	2	4	6	8	A	C	E
00	LINK	D1	D2	D3	D4	D5	D6	D7
10	D8	D9	D10	D11	D12	D13	D14	D15
20	D16	WP	PC	ST	RS	FLAGS	NONSTD	STKBLK
30	SBNDRY	STKMAX	FTOP	QLNK	PHEAP	PTASK	CODES	

2277736

Figure 8-3. The Process Record

The contents of the process record are as follows:

- LINK — Field through which all processes are circularly linked.
- D1–D16 — Display pointers to the most recently called routine at each static nesting level, 1 through 16.
- WP — Saved WP of process while not executing; address of workspace applicable to error.
- PC — Saved PC of process while not executing; address of instruction applicable to error.
- ST — Saved ST of process while not executing.
- RS — Pointer to process record of process that caused execution of this process to be resumed.
- FLAGS — Bit 0 is the heap termination flag; bits 1 through 15 are not currently used. When bit 0 is set to one and the heap space has been filled, the task terminates abnormally.
- NONSTD — Nonstandard linkage. Field contains zero except when process is executing in a routine without TIP or FORTRAN linkage. A dump of such a routine cannot be made, so when nonstandard linkage is being used, the field contains the most recent WP of a routine with TIP or FORTRAN linkage and a dump begins with the workspace pointed to by this field.
- STKBLK — Pointer to stack region of process.

- SBNDRY — Largest value reached by top of stack. Used for stack overflow checks and stack usage calculations.
- STKMAX — Address of first word beyond stack region of this process.
- FTOP — Top-of-stack pointer for FORTRAN, otherwise zero. Used to implement reentrant FORTRAN and to call TIP procedures from FORTRAN.
- GLNK — Process queue link pointer; used only in TIPMX.
- PHEAP — Pointer to the heap control block.
- PTASK — Pointer to the Task Information Block.
- CODES — Process termination code, as follows:

Code	Meaning
>FFFF	Active process
0000	Normal termination
Other value	Error code as listed in Appendix B.

8.3.4 The File Descriptor

For each FILE variable declared in a TIP routine, there is a 32-byte data block called the file descriptor, allocated in the routine stack frame. The displacement (relative address) shown in the variable map for a file variable is where the file descriptor begins. The file descriptor data block is shown in Figure 8-4.

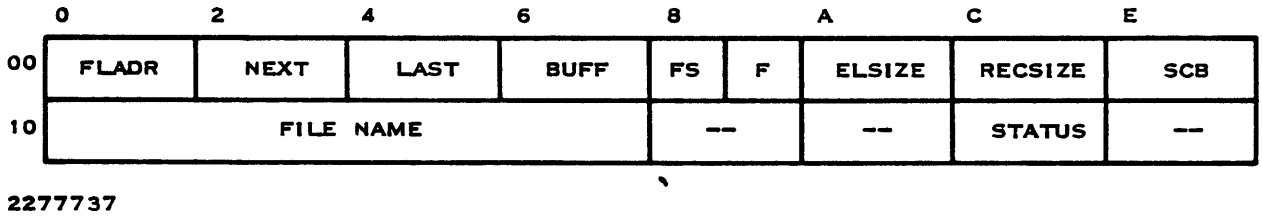


Figure 8-4. TIP File Descriptor

The contents of the TIP file descriptor are as follows:

- FLADR — Address of this file descriptor.
- NEXT — The index (based on zero) of next character in the buffer (BUFF@) to be read or written (text files only).
- LAST — The index of the last nonblank character in BUFF@ (text files only).
- BUFF — Address of buffer containing the current or next record (text and sequential files only).

- FS — File state, as follows:

  00 Closed.
  01 Open for write.
  02 Open for read.
  03 End-of-file read.
  04 End-of-medium read (during SKIPFILES).
  05 Beginning-of-medium read (during SKIPFILES).

- F — Flags, as follows:

  00.. .... Sequential.
  01.. .... Text.
  10.. .... Random.
  ....1 Terminate on any I/O error.
  ....0 Return control to user on I/O error.

- ELSIZE — Size of file element in bytes (can be odd).

- RECSIZE — Logical record length (always even).

- SCB — Address of I/O SVC block.

- FILE NAME — Logical file name as declared in TIP program.

- STATUS — Status of last I/O operation on file (returned by operating system).

The TIP file descriptor for the predefined file OUTPUT is at relative address >40 of the main program stack frame. That of the predefined file INPUT is at relative address >60 of the main program stack frame.

8.3.5 The Supervisor Call (SVC)

The interface between application programs and the operating system is the SVC. In TIP (and other high-level languages) most SVCs are provided in the run-time package for the language. However, the high-level language programmer may write SVCs to perform functions not otherwise available. Consult the *DNOS Supervisor Call (SVC) Reference Manual* and the *DNOS System Programmer's Guide* for details on using and defining SVCs.

8.3.6 I/O SVC Block

The SVC block is the data structure that defines the SVC. The structure of the I/O SVC block is shown in Figure 8-5. The I/O Operations SVC (Opcode >00) is common to all I/O operations and there is a corresponding I/O SVC block, as shown in Figure 8-5. Various sub-opcodes apply to specific types of I/O, as described below. This is not a complete definition of an I/O SVC block, but shows the features used by the TIP run-time support.

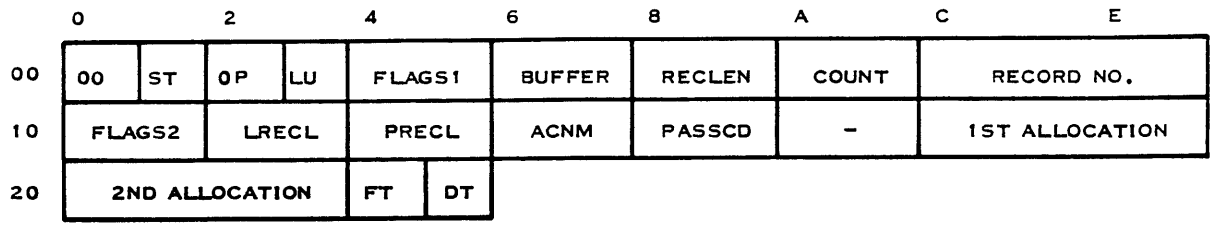


Figure 8-5. I/O SVC Block

The contents of the I/O SVC Block are as follows:

- ST — Status (returned by operating system).
- OP — Sub-opcode, as follows:

  00 Open LUNO
  01 Close LUNO
  02 Close, write EOF
  03 Open and rewind
  06 Forward space
  07 Backward space
  09 Read ASCII
  0B Write ASCII
  0D Write EOF
  0E Rewind
  12 Open extend
  91 Assign LUNO
  93 Release LUNO

- LU — Logical unit number of file.
- FLAGS1 — I/O flags and open flags:

  .1.. .... I/O error occurred
  ..1. .... End-of-file record read
  ....1 Blank Adjust
  ....0 0... Access Mode: Exclusive Write
  ....0 1... Exclusive All
  ....1 0... Shared
  ....1 1... Read only
  .... .1.. Do Not Replace Existing File

- BUFFER — Data buffer address, must be on a word boundary.
- RECLEN — Data buffer length.
- COUNT — Character count (for write operation, number of characters to be output; for read operation, set by operating system to number of characters actually input).

- RECORD NO. — Record number (relative record files only).
- FLAGS2 — Utility Flags, as follows:

1...	Created by Assign LUNO operation
.00.	File usage: Ordinary File
.01.	Directory File
.10.	Program File
.11.	Image File
...0 0...	Task local LUNO assignment
.... .1..	Generate LUNO automatically
.... ..1.	Automatic create
....0	
.... 1...	LRL flag (LRL specified in LRECL)
.... .0..	Duration: Permanent file
.... .1..	Temporary file
....0.	Disc writes may be deferred
....0 0...	Not blank suppressed
....0 1...	Blank suppressed
.... .1..	File format: Expandable
....01	Sequential
....10	Relative record
....11	Key index

- LRECL — Logical record length of file.
- PRECL — Physical record length of file.
- ACNM — Address of file access name.
- PASSCD — Address of file passcode.
- 1ST ALLOCATION — Initial file allocation.
- 2ND ALLOCATION — Secondary file allocation.

TIP extension to control block:

FT — File Type:

01FF	Sequential
02FF	Relative record
03FF	Key index
04FF	Directory
05FF	Program
06FF	Image

DT — Device Type:

0000	DUMY
0001	Hard copy terminal
0002	Line printer
0003	Cassette
0004	Card reader
0005	VDT
0006	Disk
0007	Communications
0008	Magnetic tape
8000	Symmetric IPC channel
xxFF	File (xx = file type)

8.3.7 Data Structures Used in Debugging

The stack frame, heap, process record, and file descriptor are used in debugging a TIP program. The paragraphs that deal with error termination and abnormal termination dumps in Section 9 explain the use of these data structures.

9

Debugging

9.1 INTRODUCTION

Debugging under DNOS utilizes information reported in the compiler source listing (Section 5), the link map (Section 7), the abnormal termination dump (described in this section), and the reverse assembly listing (Section 13). Debug commands allow you to execute a program in a controlled mode, temporarily suspend the program at breakpoints that you specify, view and modify the program data or instructions while the program is in memory, and then resume execution to continue testing.

The following paragraphs describe the categories of run-time errors and the resulting abnormal termination dumps, the basic SCI and Pascal debug commands, and some run-time library routines that you might encounter during debugging. Examples in this section show how to use the abnormal termination dumps (paragraph 9.2.4) and demonstrate several of the debug commands (paragraph 9.3.3).

9.2 RUN-TIME ERRORS

The following categories of errors are detected at run time and can cause abnormal termination of a TIP program:

- Failures of run-time checks specified as compiler options
- I/O errors
- Depletion of available memory space
- Task errors
- Mathematics package errors

In each case, an error message appears and an abnormal termination occurs. Error messages may appear as follows:

- An error message is displayed on the user's terminal (for a task initiated from a terminal) or in the batch stream listing (for a task executed from a batch stream). These messages are listed and explained in Appendix B. This does not apply to tasks linked with the LUNOBJ library.
- An error code number is written on the message file. These codes are listed and explained in Appendix B.
- A memory dump is written on the file OUTPUT (unless the task was linked with the MINOBJ library or the link option SPRSDUMP or NODUMP was used). For tasks linked with the LUNOBJ library, the error message precedes the dump.

- Some errors, such as storing data into the program area, might make the usual error reporting impossible. If a TIP task terminates without any error message and without writing NORMAL TERMINATION on the message file, check the system log (file .S$LOG1 or .S$LOG2) for a task error message.

9.2.1 Run-Time Checks

The following compiler options provide run-time checks by including additional code in the object module to perform checks and issue error messages:

Option	Check
CKINDEX	Array indexes within range
CKOVER	Arithmetic overflow
CKPREC	Loss of precision in FIXED or DECIMAL assignment
CKPTR	NIL pointer values
CKSET	Set elements within range
CKSUB	Subrange assignments
CKTAG	Identifiers consistent with tag fields

When any of these options has been specified for compilation of a program and the check fails at run time, abnormal termination occurs and the error message and abnormal termination dump appear. The error message includes the line number where the error occurred; this is the body line number as shown in the second column at the left of the compiler listing when the WIDELIST option is on. The name of the routine in which the error occurred is found in the dump on the file OUTPUT.

9.2.2 Memory Space Errors

The variables for a TIP program are organized in a stack (last in, first out). The variables for the main program are organized in a stack frame, and those of each routine are similarly organized in a stack frame. When executing a program, specify the amount of memory space for the stack. When execution begins, the stack frame of the main program is placed in the stack. When a routine is called, its stack frame is added to the stack. If the routine calls a nested routine, the stack frame of the nested routine is also added. Thus, at any time the stack contains the stack frame of the currently executing routine and those of its ancestors, including the main program. If a call for a routine occurs when the stack does not contain space for the routine's stack frame, an error termination occurs, displaying the following message:

```
STACK OVERFLOW
```

Memory space for dynamic variables is assigned in an area of memory called the *heap* when procedure NEW is called in a TIP program. Specify an amount of memory space for the heap when executing a program. If procedure NEW requests more memory for a variable (or record of variables) than remains in the heap and the run-time system is unable to obtain more memory space for the heap, an error termination occurs, displaying the following message:

```
HEAP FULL
```

If the requested stack and heap size are so large that the task requires a memory region greater than 65,536 bytes, the following message is displayed:

```
CANNOT GET MEMORY
```

9.2.3 Error Termination

Figure 9-1 shows the error message written on the system message file and the abnormal termination dump written on file OUTPUT for an I/O error. The termination message displayed on the terminal is as follows:

```
USH PASCAL-P527 CAN'T OPEN FILE INPUT, ACCESS NAME . INPUT14 --
                INTERNAL SVC CODE 0027
```

This indicates that the error occurred when an attempt was made to open file INPUT, access name .INPUT14. The status shown, 0027, is the completion code returned by the system; this status indicates that no file with the specified name has been defined. No further analysis of this error is necessary; assign the access name of an existing file when executing the program.

The abnormal termination dump includes a dump of each process involved in the run and a dump of the heap. The TIP runtime support system permits multiple sites of execution within the executable object code for a program. Each such site is called a *process* and has its own stack region within which its local variables are allocated. Each TIP program that executes under DNOS has at least two processes. The example program has two: DIGIO (the user program) and system process GO$. GO$ determines your stack and heap parameters, allocates and initializes both the stack region for your user program and the heap, opens the file OUTPUT, and transfers control to your user program. Upon termination of the program, GO$ again receives control and performs a controlled termination, either normal or abnormal.

The process dump consists of a dump of the process record and of the stack. The process record is the structure that describes the resources associated with each process. The fields of the process record and the hexadecimal relative byte numbers of the fields are described in Section 8.

The dump of the stack follows the dump of the process record and consists of a variable number of stack frames, each of varying length. The 128 bytes at the top of the stack (above the topmost stack frame) appear under the following heading:

```
TOP OF STACK
```

This area may contain pertinent information if another stack frame was being built when the error occurred or if the stack frame for a routine that completed remains in this area of memory. Each stack frame begins with a heading, such as the following:

```
DATA AREA FOR IO$ERR    LEVEL=2
```

In the dump, a stack frame is referred to as a *data area*. IO$ERR was executing when the dump appeared. The level, 2, is the static nesting level. The fields of the stack frame and the hexadecimal relative byte numbers of the fields are described in Section 8.

The stack frames of the program and the active routines are displayed, followed by the displays of process GO$ and any other system processes. The display for other processes is similar to that of the user process except that the area of the stack above the top stack frame is shown only for the user process.

The TIP software builds a TIP file descriptor, which describes a file to the run-time support system. The fields of the descriptor and the hexadecimal relative byte numbers of the fields are described in Section 8.

The TIP file descriptor for the predefined file OUTPUT is at relative address >40 of the main program stack frame. The file descriptor for the predefined file INPUT is at relative address >60 of the main program stack frame.

The contents of the heap follows the process dumps. This contains the dynamic variables and records for the program. The run-time system allocates file buffers and SVC blocks in the heap. Dynamic variables and records of the user program, if any, are also in the heap.

9.2.4 Using the Abnormal Termination Dump

The abnormal termination dump is useful in analyzing errors, especially when the contents of variables show what caused the error. The stack frames of the program and active routines show the progress of the program up to the point at which the error occurred. The source listing in Figure 5-3 is the listing of the program that terminated in the error of the example dump. It consists of program module DIGIO and procedure modules CCHAR and CINT. The dump of the stack includes a stack frame for DIGIO but none for CCHAR or CINT. Therefore, only DIGIO was active when the error occured; the error occurred when one of the instructions of the main program module was executed.

The information provided in the source listing (output from the compiler) as a result of specifying the MAP option is essential for examining the contents of variables. The map of identifiers for a module shows the stack frame displacement of each variable or parameter of the module. The map of identifiers for DIGIO shows that array BUFF consists of 12 bytes starting at relative address >80. In Figure 9-1, the left column contains the hexadecimal address of the first byte displayed on each line.

```
Contents of SYSMSG:          DIGIO      EXECUTION BEGINS
                             ERROR CODE = P527
                             STACK USED = 626 HEAP USED =   342

Contents of OUTPUT:

  ENTER 1 TO 5 DIGITS

         *** DUMP OF PROCESS ***

  PROCESS RECORD FOR DIGIO
  94AC (0000) 3CA4 6C2A 3D14 FF00 FF00 FF00 FF00 FF00      (<..*=..........)
  94BC (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00      (................)
  94CC (0020) FF00 6D16 172C D1CF 3CA4 8000 0000 6C28      (.....,..<......()
  94DC (0030) 6E9A 94AA 0000 0000 43E0 43BA 0527           (........C.C..   )

  TOP OF STACK
  6D92 (0000) 0527 6D82 6E5C 18E0 94AC 6D48 0000 417C      (. ...\.....H..A.)
  6DA2 (0010) 43BA 6D92 6DBE 013E 94AC 417C 23AC C1CF      (C......>..A.#...)
  6DB2 (0020) 6D7E 0000 0000 0000 0527 6D42 0000 0000      (.........  '.B....)
  6DC2 (0030) 0000 0000 0000 0000 0000 0000 0000 0000      (................)
  6DD2-6E01 SAME AS LAST LINE
  6E02 (0070) 0000 0000 0000 0000 0549 4E50 5554 0000      (.........INPUT..)
```

```
DATA AREA FOR IO$ERR        LEVEL=2
6D16 (0000) 0527 0C00 3035 3732 0027 1400 6D42 6B6B    (. ..0572. ...B..)
6D26 (0010) 6C8A 6D16 6D92 00BC 6B6E 6CE4 1C52 3B00    (..............R;.)
6D36 (0020) 0001 0000 FFFF 6D66 6C8A 0000 1449 4E50    (..............INP)
6D46 (0030) 5554 2020 203B 2E49 4E50 5554 3134 3B32    (UT    ;.INPUT14;2)
6D56 (0040) 3778 6DB0 6D48 6D4A 6D74 2580 94AC 6D16    (7....H.J..%.....)
6D66 (0050) 1DC6 C5CF 0000 FFFF 0000 6B6E 6B6E 5055    (..............PU)
6D76 (0060) 5420 2020 0000 0000 6B62 0000 942A 0000    (T    .........*..)
6D86 (0070) 0000 0800 6B6A 6D7E 0009 0000               (............    )

DATA AREA FOR OPEN$         LEVEL=2
6CE4 (0000) 0013 0013 3020 2020 0001 6D42 6C8A 6CA0    (....0    ...B....)
6CF4 (0010) 6D46 6CE4 6D16 163C 94AC 6CBA 39D2 6D3E    (.F.....<....9..>)
6D04 (0020) 0000 0000 FFFF 6D28 6C8A 0000 0014 0027    (.......(.......')
6D14 (0030) 0050                                       (.P              )

DATA AREA FOR REST$T        LEVEL=2
6CBA (0000) 0050 FFFF 0013 6BA9 3033 6D10 6C8A 0000    (.P......03......)
6CCA (0010) 6CA0 6CBA 6CE4 1B8C 94AC 6C2A 30A4 6D0C    (..........*0...)
6CDA (0020) 0000 0000 FFFF 0000 6C8A                   (..........      )

DATA AREA FOR DIGIO         LEVEL=1
6C2A (0000) 942A 0000 0000 0000 0000 0000 0000 3018    (.*............0.)
6C3A (0010) 6CEA 6C2A 6CBA 39BA 94AC 6C12 03F0 008F    (...*..9.........)
6C4A (0020) 0000 6C12 03F0 0000 0000 0000 0000 0000    (................)
6C5A (0030) 0000 0000 0000 0000 0000 0000 0000 0000    (................)
6C6A (0040) 6C6A 0011 004F 6B96 0141 0001 0050 6C00    (...4.O...A...P..)
6C7A (0050) 4F55 5450 5554 2020 0001 0100 0000 0000    (OUTPUT  ........)
6C8A (0060) 6C8A 0000 0000 0000 0041 0002 0000 6B6E    (.........A......)

6C9A (0070) 494E 5055 5420 2020 0001 0100 0027 0000    (INPUT    .....'..)
6CAA (0080) 0000 0000 0000 0000 0000 0000 0000 0000    (................)
```

Figure 9-1. Abnormal Termination Dump (Sheet 1 of 2)

```
        *** DUMP OF PROCESS***

PROCESS RECORD FOR GO$
3CA4 (0000) 94AC 6C2A 3DC2 FF00 FF00 FF00 FF00 FF00    (...*=..........)
3CB4 (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00    (................)
3CC4 (0020) FF00 3D14 28F6 FFFF 94AC 8000 0000 3CE6    (..=.(.........<.)
3CD4 (0030) 3FC4 4172 0000 0000 43E0 43BA FFFF         (?.A.....C.C...  )

DATA AREA FOR TERM$         LEVEL=2
3D14 (0000) 40F2 0000 0027 6C6A 3DB4 43BA 6C2A 0005    (@....'..=.C..*..)
3D24 (0010) 27A8 3D14 3D88 0636 3CA4 3CE8 13C2 3DB0    ('.=.=..6<.<...=.)
3D34 (0020) FF00 3CE8 FFFF 0000 94AC 0000 0012 4749    (..<...........GI)
3D44 (0030) 0005 0027 6C6A 002D 4543 5554 494F 4E20    (...'...-ECUTION )
3D54 (0040) 4245 4749 4E53 2E28 010E 0009 4552 524F    (BEGINS.(....ERRO)
3D64 (0050) 5220 434F 4445 203D 2050 3532 37F0 43BA    (R CODE = P527.C.)
3D74 (0060) 3D62 3DE4 2D5C 3CA4 3D14 160C D1CF 3D14    (=.=.-\<.=.....=.)
3D84 (0070) 3D14 1600                                  (=...            )
```

```
DATA AREA FOR GO$           LEVEL=2
3CE8 (0000) 40F2 0000 0000 0000 0000 00000000 94AC      (@..............)
3CF8 (0010) 0000 3CE8 3D14 264E 3CA4 3CD0 03F0 3D3C     (..<.=.&N<.<...=<)
3D08 (0020) 0000 0000 FFFF 0000 94AC 94AC               (............    )
*** END OF PROCESS DUMP

          *** DUMP OF HEAP ***

3C14 (0000) 4552 524F 5220 434F 4445 203D 2050 3532     (ERROR CODE  = P52)
3C24 (0010) 3720 4E20 4245 4749 4E53 0000 0000 0000     (7 N BEGINS......)
3C34 (0020) 0000 0000 0000 0000 0000 0000 0000 0000     (................)
3C44-3C53  SAME AS LAST LINE
3C54 (0040) 0000 0000 0000 0000 0000 0000 0000 0000     (................)

3C66 (0000) 1244 5046 494C 452E 4558 5052 4F47 2E4D     (.DPFILE.EXPROG.M)
3C76 (0010) 5353 4700

3C7C (0000) 0000 0B01 0001 3C14 0050 0012 0000 0000     (......<..P......)
3C8C (0010) 868D 0000 0000 3C66 0000 0000 0000 0000     (......<.........)
3C9C (0020) 0000 0000 01FF                              (......          )

6B62 (0000) 082E 494E 5055 5431 3420                    (..INPUT14       )

6B6E (0000) 0027 9104 4018 0000 0050 0000 0000 0000     (.'..@....P......)
6B7E (0010) 0409 0000 0000 6B62 0000 0000 0000 0000     (................)
6B8E (0020) 0000 0000 0000                              (......          )

6B96 (0000) 2036 4239 3620 2830 3030 3029 2032 3033     ( 6B96 (0000) 203)
6BA6 (0010) 3620 3432 3332 2033 3333 3220 3230 3333     (0 3432 3332 2033)
6BB6 (0020) 2033 3333 3320 3332 3230 2033 3233 3320     ( 3230 2033 3233 )
6BC6 (0030) 3333 3230 2020 2820 3332 3330 2032 3033     (3033  ( 3230 203)
68D6 (0040) 3233 3020 3230 3329 0000 0000 0000 0000     (230 203).......)

6BE8 (0000) 1444 5046 494C 452E 4558 5052 4F47 2E4F     (.DPFILE.EXPROG.O)
6BF8 (0010) 5554 5055 5400                              (UTPUT.          )

6C00 (0000) 0000 0B02 0001 3E18 0050 0002 0000 0000     (......>..P......)
6C10 (0010) 868D 0000 0000 6BE8 0000 0000 0000 0000     (................)
6C20 (0020) 0000 0000 01FF                              (......          )
```

Figure 9-1. Abnormal Termination Dump (Sheet 2 of 2)

The number in parentheses is the hexadecimal displacement of that byte from the start of the routine's stack frame (data area). The 12 bytes in relative bytes >80 through >8F of the stack frame of DIGIO contain zeros. The map of identifiers shows that variable I occupies 2 bytes at relative byte >8C. This location also contains zero. Variable NUM occupies 2 bytes at relative byte >8E, which also contains zero. If the error involved the data in any of these variables, locating the values of the variables in the stack frame enables you to determine the cause of the error.

The variables of routines may be parameters passed by value or parameters passed by reference. In the former case, the stack frame contains the value of these parameters when the routine was called. In the latter case, the stack frame contains the address of the parameter.

Information in the stack frames is useful in determining the point at which the error occurred. To trace execution back to the instruction that failed in the user program requires the source listing (Figure 5-3), the abnormal termination dump (Figure 9-1), the link map (Figure 9-2), and the reverse assembly listing (Figure 13-1).

```
 LINKER        1.2.0  82.237            01/07/84  *11:08:04          PAGE            1
COMMAND LIST
FORMAT IMAGE,REPLACE
LIBRARY .TIP.OBJ
TASK DIGIO
INCLUDE (MAIN)
INCLUDE DPFILE.EXPROG.EXOBJ
END

 LINKER        1.2.0  82.237            01/07/84  11:08:04           PAGE            2
LINK MAP
CONTROL FILE = DPFILE.EXPROG.LC

LINKED OUTPUT FILE = DPFILE.EXPROG.EXPROG

LIST FILE = DPFILE.EXPROG.EXLMAP
                                                   NUMBER OF OUTPUT RECORDS = 65
OUTPUT FORMAT = IMAGE

LIBRARIES

NO    ORGANIZATION  PATHNAME

1     RANDOM        .TIP.OBJ

 LINKER        1.2.0  82.237            01/07/84  11:08:04           PAGE    3

PHASE 0, DIGIO       ORIGIN = 0000   LENGTH = 4416    (TASK ID = 1)

MODULE     NO     ORIGIN    LENGTH      TYPE          DATE        TIME       CREATOR

MAIN        1       0000      2F34     INCLUDE,1     12/16/83    21:32:05    SDSLNK
$DATA       1       3BD4      05DC
CCHAR       2       2F34      0076     INCLUDE       01/07/84    11:03:51    DXPSCL
CINT        3       2FAA      005C     INCLUDE       01/07/84    11:03:52    DXPSCL
DIGIO       4       3006      0126     INCLUDE       01/07/84    11:03:55    DXPSCL
SCIRTNS     5       312C      0770     LIBRARY,1     12/16/83    21:30:41    SDSLNK
$DATA       5       41B0      020A
INIT$1      6       389C      005A     LIBRARY,1     12/16/83    18:07:43    DXPSCL
P$INIT      7       38F6      0002     LIBRARY,1     12/16/83    19:52:33    SDSMAC
PB$INIT     8       38F8      0002     LIBRARY,1     12/16/83    19:52:45    SDSMAC
MESAG$DX    9       38FA      006E     LIBRARY,1     12/16/83    18:07:52    DXPSCL
P$TERMDP    10      3968      0042     LIBRARY,1     12/16/83    21:06:33    SDSMAC
MM$DIRDU    11      39AA      0002     LIBRARY,1     12/16/83    19:51:04    SDSMAC
REST$T      12      39AC      004E     LIBRARY,1     12/16/83    18:21:10    DXPSCL
GET$CH      13      39FA      002E     LIBRARY,1     12/16/83    19:49:26    SDSMAC
RDLN$       14      3A28      00CC     LIBRARY,1     12/16/83    18:20:42    DXPSCL
GET$RCOR    15      3AF4      00E0     LIBRARY,1     12/16/83    18:19:36    DXPSCL
```

Figure 9-2. Link Map of Example Program (Sheet 1 of 2)

```
COMMON     NO    ORIGIN    LENGTH

CUR$       1      43BA     0026
HEAP$      1      43E0     0010
SYS$MS     9      43F0     0020
PARM$      1      4410     0006

                              D E F I N I T I O N S

  NAME    VALUE    NO  *NAME    VALUE    NO   NAME    VALUE    NO   NAME    VALUE    NO

*ABEND$   009C     1   *ABND$0  00C4     1   *ABND$1  00BC     1   *ABND$2  008E     1
*BID$SV   2B00*    1    CCHAR   2F46     2    CINT    2FB8     3   *CLOSE$  0160     1
 CLS$     01C0     1   *CLS$FI  0222     1   *CMP$ST  0250     1   *CREAT$  0272     1
*CUR$     03A8     1   *CUR$$   0EDA     1    DIV$    03B6     1   *DMPP$H  043C     1
*DSTR$$   03F0     1   *ENC$T   0B72     1   *ENI$T   0C28     1   *ENS$T   0D60     1
*ENT$     0E42     1    ENT$1   0060     1    ENT$2   0E32     1    ENT$M   0E4C     1
 ENT$MD    0EF6     1    ENT$S   0ECA     1   *ENX$T   0F1A     1    EOLN$   108C     1
*FIND$S   2A58     1    FL$INI  11C8     1   *FREE$   10A6     1    GET$CH  39FA    13
 GET$RC    3B10    15   *HEAP$T  13D6     1    INIT$1  38B8     6    IO$ERR  163C     1
*MAP$     174E     1   *MARG$N  0080     1    MESAG$  390A     9    MM$DIR  39AA    11
 MOV$4     18AE     1   *MOV$5   18AC     1    MOV$6   18AA     1   *MOV$7   18A8     1
*MOV$8    18A6     1   *MOV$N   18A0     1   *MSG$    18B8     1   *NEW$    18E0     1
 OPEN$     1B8C     1   *OPN$FI  1CEE     1   *P$ABND  0138     1    P$INIT  38F6     7
 P$TERM    3976    10   *PATCH$  0E62     1    PB$INI  38F8     8
*PM$CHK   0000*    11   *PM$CMP  0000*    11  *PM$FLT  0000*    11   *PM$FTN  0000*    11
*PM$IO    0000*    11   *PM$MIS  0000*    11  *PM$MPK  0000*    11   *PM$TIO  0000*    11
*PRT$ME   1EEA     1    PSCL$$   303E     4   *PUT$RC  220C     1    PUTCH$  21BE     1
 RDLN$     3A42    14    REST$T   39BA    12   *RESUM$  23A8     1    RET$1   006E     1
 RET$2     0EE0     1    RET$M   0EF2     1    RET$S   0EF4     1    REWND$  22DA     1
 LINKER             1.2.0  82.237              01/07/84    11:08:04              PAGE    4
 NAME    VALUE    NO   NAME    VALUE    NO   NAME    VALUE    NO   NAME    VALUE    NO
 REWRT$  2344     1   *RSUMR$  23E4     1    S$GTCA  4340     5   *S$IADD  41B0     5
*S$IASC   41C4     5   *S$IDIV  41F4     5   *S$IMUL  4200     5    S$INT   420C     5
*S$ISUB   41B8     5    S$MAPS  423C     5   *S$NEW   424A     5    S$PARM  427C     5
 S$PTCA    4352     5   *S$RTCA  4356     5   *S$SCPY  428E     5    S$SETS  42A2     1
*S$STOP   42B2     5    S$TERM  42AE     5   *SET$NA  252C     1   *STACK$  2556     1
*STORE$   2A30     1    SVC$    257A     1    T$CC    000A*    1    T$EC    000C*    1
 T$ES      001E*    1   *T$INID  0004*    1   *T$MDIR  0020*    1    T$MN    001A*    1
 T$MSG     000E*    1   *T$RNID  0003*    1   *T$STN   0005*    1   *T$TERM  0022*    1
*T$TIB    43BA     1    T$VT    001C*    1   *TX$ERR  2A88     1   *WRC$T   2B2A     1
*WREOF$   2C02     1   *WRI$T   2C82     1    WRLN$   2D5C     1    WRS$T   2DC8     1
*WRX$T    2E6A     1

**** LINKING COMPLETED
```

Figure 9-2. Link Map of Example Program (Sheet 2 of 2)

Relative address >24 (PC) of the DIGIO process record contains the contents of the PC when process GO$ received control for termination. In the example, this address is >172C. From the link map (Figure 9-2) the code at that address is found to be part of module MAIN, loaded at address 0000 and extending through address >2F34. Since the top stack frame in the stack is that of IO$ERR, the PC address in the process record should be in that routine. To confirm this, it would be necessary to check the link map for the partial link of the MAIN module. However, the routines in MAIN are included in alphabetical order, so in this particular case you can consult the definitions section of the DIGIO link map. In other cases, consult the link map for the partial link if you do not know the order that the routines are included.

The link maps for the partial links are in files on the installation disk under the directory DNPASCAL.TIP.LIST. The files and the link maps they contain are as follows:

File Name	Link Map
LUNMAINL	.TIP.LUNOBJ.MAIN
SCIMAINL	.TIP.OBJ.MAIN
SHRPROCL	.TIP.OBJ.SHRPROC

The definitions section of the DIGIO link map shows the entry point of routine IO$ERR to be >163C and the entry point of the routine above IO$ERR to be >174E (MAP$), so the PC value stored is from routine IO$ERR.

Check the stack frames of OPEN$ and REST$T to verify the path of control through these routines. Relative address >1C contains >30A4, indicating that control passed to routine REST$T from that point. The module list of the link map shows that the main module of DIGIO was loaded at address >3006 and occupies >126 bytes. Address >30A4 is address >9E relative to the load point >3006. The reverse assembly listing (Figure 13-1) shows that relative address >9E is the address of an INC instruction that follows a BLWP instruction. That BLWP instruction transferred control to routine REST$T.

The events resulting in the termination are as follows: DIGIO calls REST$T, implementing the RESET procedure call in the source program. REST$T calls routine OPEN$ to open file INPUT. The open operation fails because the file assigned for input has not been created. OPEN$ calls routine IO$ERR. IO$ERR passes control to process GO$, which terminates the program and prints the dump.

9.2.5 Unformatted Abnormal Termination Dump

The linking procedures describe the linking of an alternate abnormal termination dump routine by the command INCLUDE (SPRSDUMP). Figure 9-3 shows an example of the unformatted dump written by this routine. The dump contains approximately the same information, printed in the order of memory address rather than in the order of the formatted abnormal termination dump (process record, stack region, heap region). The four characters printed in the upper left corner of the unformatted dump specify the address of the process record. The left column contains the hexadecimal addresses of the first words displayed on the lines of the dump. The numbers in parentheses are relative addresses within the blocks of data displayed in the dump. The next eight columns show the contents of eight words represented as hexadecimal numbers. The right column contains the contents of the same eight words as ASCII characters, with unprintable characters represented by periods (.). A group of three periods at the left on a line represents one or more lines identical to the preceding line (normally containing zeros).

With the process record address, you can find the process record in the dump. The contents of the process record are listed in Section 8. The more useful items of information in the process record are as follows:

- Address of the stack frame of the main program — relative address >0002

- Contents of WP — Relative address >0022
- Contents of PC — Relative address >0024
- Address of the beginning of the stack region — Relative address >002E
- Termination code (listed in paragraph B.2) — Relative address >003C

To use the unformatted dump, note the address of the process record and locate the block that contains the process record. The DISPLAY array shows the address of the stack frame for the most recently called routines at each static nesting level. The contents of each stack frame are described in Section 8. The compiler source listing (when written with the MAP option) shows the stack frame relative address of each variable. Pointers in the stack frame identify stack frames of other routines that were called at each static nesting level. File descriptors that describe the files defined for the task are also located in the stack frame, as described in Section 8.

```
ENTER 1 TO 5 DIGITS

88EC

 304A(0000) 4552 524F 5220 434F 4445 203D 2050 3532    ERROR CODE = P52
 305A(0010) 3720 4E20 4245 4749 4E53 0000 0000 0000    7 N BEGINS......
 306A(0020) 0000 0000 0000 0000 0000 0000 0000 0000    ................
...
 308A(0040) 0000 0000 0000 0000 0000 0000 0000 0000    ................

 309C(0000) 1344 5046 494C 452E 4558 5052 4F47 322E    .DPFILE.EXPROG2.
 30AC(0010) 4D53 4732                                            MSG2

 30B2(0000) 0000 0B01 0001 304A 0050 0012 0000 0000    ......0J.P......
 30C2(0010) 068D 0000 0000 309C 0000 0000 0000 0000    ......0.........
 30D2(0020) 0000 0000 01FF                                      ......

 30DA(0000) 88EC 606A 31EE FF00 FF00 FF00 FF00 FF00    ....1...........
 30EA(0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00    ................
 30FA(0020) FF00 311E 0584 D1CF 88EC 8000 0000 311C    ..1...........1.
 310A(0030) 33FA 35A8 0000 0000 3816 37F0 FFFF 0000    3.5.....8.7.....

 311C(0000) 055C 3528 0000 0000 0000 0000 0000 0000    .\5(............
 312C(0010) 88EC 0000 311E 314A 0714 30DA 3106 0466    ....1.1J..0.1...
 313C(0020) 3172 0000 0000 FFFF 0000 88EC 88EC 3528    1.............5(
 314C(0030) 0000 0027 60AA 31EA 37F0 606A 0005 086E     ...'..1.7.......
 315C(0040) 314A 31BE 00BC 30DA 311E 0598 31E6 FF00    1J1...0.1...1...
 316C(0050) 311E FFFF 0000 88EC 0000 0012 4749 0005    1...........GI..
 317C(0060) 0027 60AA 002D 4543 5554 494F 4E20 4245     .'...-ECUTION BE
 318C(0070) 4749 4E53 2E5A 010E 0009 4552 524F 5220    GINS.Z....ERROR
 319C(0080) 434F 4445 203D 2050 3532 3726 37F0 3198    CODE = P527&7.1.
 31AC(0090) 321A 0AE4 30DA 314A 10DA D1CF 314A 314A    2...0.1J....1J1J
 31BC(00A0) 10CE 0001 300A 35AA 311C 35A8 0491 35A8    ....0.5.1.5...5.
 31CC(00B0) 300A 3816 31BE 31EE 0146 60AA 314A 09C6    0.8.1.1..F..1J..
 31DC(00C0) D1CF FF00 314A 09C6 FFFF 88EC 0001 0000    ....1J..........
 31EC(00D0) 3176 3176 31F2 35AA 31FC 31EC 00DC 0000    1...1.5.1.1.....
```

Figure 9-3. Unformatted Abnormal Termination Dump (Sheet 1 of 3)

```
31FC(00E0) 2000 5FE8 31EE 321E 019A 60AA 31BE 0000    ....1.2.....1...
320C(00F0) 017C 31BE 31BE 0132 0001 311C 35A8 0050    ..1.1..2..1.5..P
321C(0100) FFFF 001A 60C6 0000 6040 0000 3276 60AA    .........@..2...
322C(0110) 0000 2598 321E 3248 1D66 30DA 31EE 01DC    ..%.2.2H..0.1...
323C(0120) 31CF FFFF 6040 FFFF 0002 60AA 2020 3298     1....@......2.
324C(0130) 6040 0000 60C6 0000 6040 0048 30DA 3248    .@.......@.H0.2H
325C(0140) 327A 2598 30DA 321E 0B12 31CF C1CF 0011    2.%.0.2...1.....
326C(0150) FFFF 0000 6040 5FD8 0048 60C6 0012 5450    .....@_..H....TP
327C(0160) 6040 329C 2598 30DA 3240 0B12 31CF 32A0    .@2.%.0.2@..1.2.
328C(0170) 327A 32A4 259E 30DA 3248 1E12 D1CF 0000    2.2.%.0.2H......
329C(0180) 3362 0000 30DA 6040 2E8C C5CF 01FF 0000    3...0..@........
32AC(0190) FFFF 329C 32C6 259E 30DA 326A 1E12 D1CF    ..2.2.%.0.2.....
32BC(01A0) 0006 0013 0000 5046 30B2 452E 4558 5052     .....PF0.E.EXPR
32CC(01B0) 4F47 322E 4F55 5432 534D 5347 2020 0000    OG2.OUT2SMSG   ..
32DC(01C0) 0000 309C 0000 3528 0000 0000 0000 30AF    ..0...5(......0.
32EC(01D0) 32F5 0014 0000 0000 32E2 33BC 2600 30DA    2.......2.3.&.0.
32FC(01E0) 32A8 2E8C C5CF 0000 0000 FFFF 0000 5359    2.............SY
330C(01F0) 534D 5347 004F 5554 5055 5406 0013 1344    SMSG.OUTPUT....D
331C(0200) 5046 494C 452E 4558 5052 4F47 322E 4D53    PFILE.EXPROG2.MS
332C(0210) 4732 0000 0000 0000 0000 0000 0000 0000    G2..............
333C(0220) 0000 0000 0000 0000 0000 0000 0000 0000    ................
...
335C(0240) 0000 0000 0013 0000 32C0 3310 005C 3284    ........2.3..\2.
336C(0250) 602A 4D53 0001 0000 3362 33A0 2606 30DA    .*MS....3.3.&.0.
337C(0260) 3288 2F64 C5CF 314A 3288 FFFF 0000 3284    2./...1J2.....2.
338C(0270) 0016 6028 603E 3866 0000 0000 27C2 0000    ...(.>8.....'...
339C(0280) 0000 0001 0000 0000 0000 0000 0000 0000    ................
33AC(0290) 0000 0000 0000 0000 0000 0000 0000 0013    ................
33BC(02A0) 3528 331A 336A 0014 32DE 309C 0000 0001    5(3.3...2.0.....
33CC(02B0) 0000 33BC 33FA 2606 30DA 32E2 2F64 C5CF    ..3.3.&.0.2./...
33DC(02C0) 3198 32E2 FFFF 0000 32DE 0016 309A 30B0    1.2.....2...0.0.
33EC(02D0) 3010 0000 0000 008A 0000 0000 0001 0000    0...............
33FC(02E0) 0000 0000 0000 0000 0000 0000 0000 0000    ................
...
359C(0480) 0000 0000 0000 0000 0000 0000 0000          ..............

5FA4(0000) 082E 494E 5055 5431 3420                   ..INPUT14

5FB0(0000) 0027 9104 4018 0000 0050 0000 0000 0000    .'..@....P......
5FC0(0010) 0409 0000 0000 5FA4 0000 0000 0000 0000    ......_.........
5FD0(0020) 0000 0000 0000 5FA4 0000 0000 0000 0000    ......

5FD8(0000) 2035 4644 3828 3030 3030 2920 3230 3335      5FD8(0000) 2035
5FE8(0010) 2034 3634 3420 3334 3230 2033 3333 3420       3634 3420 3334
5FF8(0020) 3332 3330 2032 3033 3320 3333 3333 2033      3320 3333 3333 2
6008(0030) 3033 3320 2020 2033 3332 3020 3333 3333     333    333      33
6018(0040) 3320 2020 2033 3320 0000 0000 0000 0000     3      33 ........

602A(0000) 1344 5046 494C 452E 4558 5052 4F47 322E    .DPFILE.EXPROG2.
603A(0010) 4F55 5432                                  OUT2
```

Figure 9-3. Unformatted Abnormal Termination Dump (Sheet 2 of 3)

```
6040(0000) 0000 0B02 0001 3248 0050 0002 0000 0000     ......2H.P......
6050(0010) 068D 0000 0000 602A 0000 0000 0000 0000     ........*........
6060(0020) 0000 0000 01FF                              ......

6068(0000) 0378 886A 0000 0000 0000 0000 0000 0000     ................
6078(0010) 0352 612A 606A 60FA 0D22 88EC 6052 0466     .R.*.....".....R..
6088(0020) 008F 0000 6052 0466 0000 0000 0000 0000     .....R..........
6098(0030) 0000 0000 0000 0000 0000 0000 0000 0000     ................
60A8(0040) 0000 60AA 0016 004F 5FD8 0141 0001 0050     .....<.0_..A...P
60B8(0050) 6040 4F55 5450 5554 2020 0001 0100 0000     .@OUTPUT  ......
60C8(0060) 0000 60CA 0000 0000 0000 0041 0002 0000     ...........A....
60D8(0070) 5FB0 494E 5055 5420 2020 0001 0100 0027      _.INPUT     .....'
60E8(0080) 0000 0000 0000 0000 0000 0000 0000 0000     ................
60F8(0090) 0000 0050 FFFF 0013 5FEB 036D 6150 60CA     ...P...._....P..
6108(00A0) 0000 60E0 60FA 6124 2080 88EC 606A 03DE     .......$ .......
6118(00B0) 614C 0000 0000 FFFF 0000 60CA 0013 0013     .L..............
6128(00C0) 035A 2020 0001 6182 60CA 60E0 6186 6124     .Z    ..........$
6138(00D0) 6156 1E4E 88EC 60FA 0D3A 617E 0000 0000     .V.N.....:......
6148(00E0) FFFF 6168 60CA 0000 0014 0027 0050 0527     ............'.P.'
6158(00F0) 0C00 3035 3732 0027 1400 6182 5FAD 60CA     ..0572. '...._...
6168(0100) 6156 61D2 0E38 5FB0 6124 2146 3B00 0001     .V...8_..$!F;...
6178(0110) 0000 FFFF 61A6 60CA 0000 1449 4E50 5554     ...........INPUT
6188(0120) 2020 203B 2E49 4E50 5554 3134 3B32 37B8        ;.INPUT14;27.
6198(0130) 61F0 6188 618A 61B4 259E 88EC 6156 2C34     ........%....V,4
61A8(0140) C5CF 0000 FFFF 0000 5FB0 5FB0 5055 5420     ..........__.PUT
61B8(0150) 2020 0000 0000 5FA4 0000 886A 0000 0000        ...._.........
61C8(0160) 0800 5FAC 61BE 0009 0000 0527 61C2 629C     .._..... '....
61D8(0170) 2600 88EC 6188 0000 35B2 37F0 61D2 61FE     &.......5.7.....
61E8(0180) 0EBA 88EC 35B2 10F0 C1CF 61BE 0000 0000     ....5...........
61F8(0190) 0000 0527 6182 0000 0000 0000 0000 0000     ... '...........
6208(01A0) 0000 0000 0000 0000 0000 0000 0000 0000     ................
...
6248(01E0) 0000 0549 4E50 5554 0000 0000 0000 0000     ...INPUT........
6258(01F0) 0000 0000 0000 0000 0000 0000 0000 0000     ................
...
6298(0230) 0000 0000 886A 624A 61FA 00A4 61BE 5FA4     .......J...._..
62A8(0240) 0000 0001 0000 629C 62DA 2606 88EC 61C2     ..........&.....
62B8(0250) 2F96 C1CF 314A 61C2 FFFF 0000 61BE 000C     /...1J..........
62C8(0260) 5FA2 5FAE 3866 0000 0000 273C 0000 0000     __.8.....  '<....
62D8(0270) 0001 0000 0000 0000 0000 0000 0000 0000     ................
62E8(0280) 0000 0000 0000 0000 0000 0000 0000 0000     ................
...
88E8(2880) 0000

88EC(0000) 30DA 606A 314A FF00 FF00 FF00 FF00 FF00     0...1J..........
88FC(0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00     ................
890C(0020) FF00 6156 1F3E D1CF 30DA 8000 0000 6068     ...V.>..0.......
891C(0030) 62DA 88EA 0000 0000 3816 37F0 0527           ........8.7..'
```

Figure 9-3. Unformatted Abnormal Termination Dump (Sheet 3 of 3)

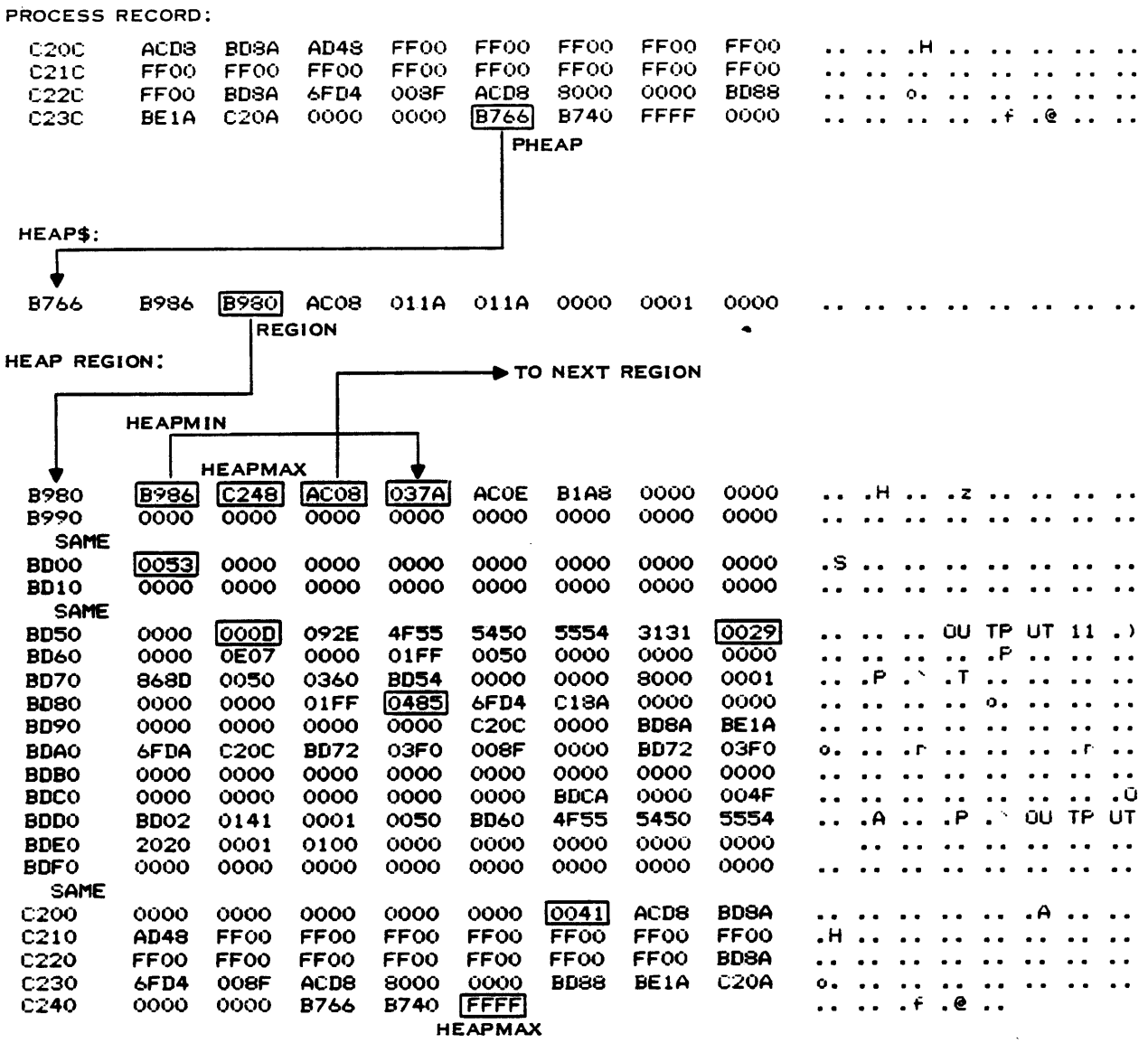
9.2.6 Debugging Heap Errors

The following errors might indicate invalid data in the heap:

- P105 ADDRESSING ERROR
- P365 INVALID PACKET POINTER FOR DISPOSE
- P366 INVALID LENGTH OF PACKET IN DISPOSE
- P405 INVALID RELEASE POINTER

Some heap memory errors can result from overwriting part of the packet list header with data from the preceding packet. For example, a dynamic array initialized in a packet and indexed beyond the packet might write into the succeeding packet list header. When this happens, the length field and perhaps the predecessor and successor fields of the packet list header become invalid. Check for negative values and values that are beyond the size of the heap region. Also, the predecessor and successor fields should form a circularly linked chain through the heap structure. (Section 8 describes the heap structure.)

Figure 9-4 shows the pointers from the process record to HEAP$, and from HEAP$ to the header block of the heap region. HEAPMIN points to the first packet in the region. Use the length words to find each packet in the region.




```
FREE PACKET AT B986, LENGTH 037A
                B986+037A=BD00
ALLOCATED PACKET AT BD00, LENGTH 0052
ALLOCATED PACKET AT BD52, LENGTH 000C
ALLOCATED PACKET AT BD5E, LENGTH 0028
ALLOCATED PACKET AT BD86, LENGTH 0484
ALLOCATED PACKET AT C20A, LENGTH 0040
```

Figure 9-4. Example Heap Region

By checking the length, and the predecessor and successor fields, you might recognize the incorrect data and determine the origin of the error. Otherwise, you can set a series of breakpoints in the program. By checking the memory contents at each point, you should be able to identify the part of the program that generates the error.

9.3 DEBUG COMMANDS

The DNOS and TIP software allows debugging with both SCI and Pascal debug commands. The following paragraphs describe the Pascal debug commands and six SCI debug commands. (The *DNOS System Command Interpreter (SCI) Reference Manual* describes the remaining SCI debug commands.) Paragraph 9.3.3 demonstrates the use of several debug commands.

Use the Pascal debug commands to debug tasks linked in any of the ways described in Section 7.

The Pascal debug commands provide breakpoint capability and the ability to display the Pascal stack. Symbolic debugging is not supported in Pascal tasks.

The following limitations apply to a task to be debugged using the Pascal debug commands:

- The stack display commands SPS and LPS cannot be used while the TIP task is executing in a FORTRAN routine linked in the task or in any routine with nonstandard linkage. (Section 11 describes the standard interface for assembly language routines.)

- To reference a Pascal routine by name or to display the name, the routine must have been compiled with the TRACEBACK option (a default option) in effect.

Both the SCI and Pascal debug commands require that the task be unconditionally suspended either before or during the execution of the command. In the unconditionally suspended state (task state 6), the task is suspended until activated by a command. A task may be unconditionally suspended in any of the following ways:

- The Pascal task is executed with the debug mode selected, which loads the task for execution but suspends it before execution actually begins.

- The task suspends itself (for example, by calling the library procedure SUSPEND).

- The task executes a breakpoint (XOP 15, 15).

Once the task is suspended, use debug commands to assign breakpoints, simulate execution, display memory, and perform other debug functions. When finished debugging, enter a Kill Background Task (KBT) command to terminate the task.

To place the task in the controlled mode, execute the Execute Debug (XD) command. Placing a Pascal task in the controlled mode enables displays associated with breakpoints and allows you to enter the controlled mode debug commands. Otherwise, you must monitor the status of the task by using the Show Pascal Stack (SPS) command or the Show Internal Register (SIR) command.

When a VDT is executing the Pascal task in controlled mode, a breakpoint causes a display of data. When the terminal is a TTY device or the VDT is in the TTY mode, no display occurs.

9.3.1 Basic SCI Debug Commands

The basic SCI debug commands include the Debug control commands Execute Debug (XD) and Quit Debug (QD), the task control commands Halt Task (HT) and Resume Task (RT), and the data modification commands Modify Memory (MM) and List Memory (LM). The *DNOS System Command Interpreter (SCI) Reference Manual* describes other debug commands. You can enter the commands when the SCI command prompt ([ ]) appears.

Regarding the response type indicators for certain Debug command prompts, an integer expression can be a decimal or hexadecimal value or an expression composed of decimal or hexadecimal integers and the operators +, −, *, and /. A full expression refers to an integer expression with the additional operators <, >, and ( ). String operands and the symbols #PC, #WP, #ST, and #Rn are also permitted in the SCI Debug controlled mode. An expression list is a series of expressions separated by commas.

9.3.1.1 Execute Debug (XD). This command places the specified task into controlled mode. The run-time ID is optional and cannot be that of a system task. If no run-time ID is given, an automatic call is made to the Execute and Halt Task (XHT) command to place the task into execution. The parameter SYMBOL TABLE OBJECT FILE does not apply to Pascal tasks and should be left blank.

The Debugger can simulate 990/12 object code (when executing on a 990/12) or 990/10 object code (when executing on a 990/10 or a 990/12). The command defaults to the object code of the type of the host computer. Only one task for each station may be in debug mode at a given time.

Syntax:

```
[]XD
EXECUTE DEBUG

                  RUN ID:  integer  (*)
SYMBOL TABLE OBJECT FILE:  [pathname@]
     990/12 OBJECT CODE?:  {Y/N}    (YES)
```

Example:

```
EXECUTE DEBUG
                  RUN ID:  >9A
SYMBOL TABLE OBJECT FILE:
     990/12 OBJECT CODE:  N
```

9.3.1.2 Quit Debug (QD). This command takes a task out of debug or controlled mode. You have the option of killing the task at this point. If you choose not to kill the task, it remains unconditionally suspended. You can still issue any of the general SCI commands. Use the Resume Task (RT) or Proceed from Breakpoint — Pascal (PBP) command (depending on whether the task is at a breakpoint) to activate the task.

Syntax:

```
[] QD
QUIT DEBUG MODE
KILL TASK?:  {Y/N}    (YES)
```

Example:

```
QUIT DEBUG MODE
      KILL TASK?:  Y
```

9.3.1.3 Halt Task (HT). The specified task is unconditionally suspended at the end of the current time slice. Only a privileged user can halt a system task. If the specified task is already unconditionally suspended, this command is turned into a no operation command. If the task is not in the active state, this command waits five seconds for the task to reach the unconditionally suspended state and then gives you the option of aborting or continuing to wait. This occurs every five seconds.

Syntax:

```
[] HT
HALT TASK
   RUN ID:  integer expression ( )
```

Example:

```
HALT TASK
   RUN ID:  >7
```

If the task cannot be suspended, the following message appears:

```
UNABLE TO SUSPEND TASK. CURRENT STATE=XX.CONTINUE COMMAND?
```

If you enter a YES response, another attempt is made to suspend the task. If the attempt is unsuccessful, the message appears again. A NO response to the preceding message causes the following message to appear:

```
DO YOU WISH TO LEAVE SUSPENSION PENDING?
```

A YES response leaves the suspension pending, while a NO response terminates the suspension attempt.

9.3.1.4 Resume task (RT). The specified task is activated at the point at which it was suspended. The run-time ID is assigned when the task is executed. The specified task must be unconditionally suspended when this command is executed or an error occurs. Use either the Delete Breakpoint — Pascal (DBP) and the RT commands, the Proceed from Breakpoint — Pascal (PBP) command, or the Delete and Proceed from Breakpoint — Pascal (DPBP) command to restart the task halted at a breakpoint. To reactivate a halted task, use the RT command rather than the Activate Task (AT) command.

Syntax:

```
[] RT
RESUME TASK
     RUN ID:  integer expression ( )
```

Example:

```
RESUME TASK
     RUN ID:  >7
```

9.3.1.5 List Memory (LM). This command lists the specified memory area of a program on the specified output device or file. The output defaults to the terminal. If the task is not unconditionally suspended, it is temporarily suspended while the listing is being formatted.

Syntax:

```
        [] LM
        LIST MEMORY
             RUN ID:  integer expression ( )
   STARTING ADDRESS:  full expression
    NUMBER OF BYTES:  [full expression]  (length of display) (*)
LISTING ACCESS NAME:  [pathname@]  (output device or file name) (*)
```

Example:

```
         LIST MEMORY
             RUN ID:  >80
   STARTING ADDRESS:  >102
    NUMBER OF BYTES:  >14A
LISTING ACCESS NAME:
```

9.3.1.6 Modify Memory (MM). The MM command allows you to modify the memory image of a task, starting at a specified address. The contents of the specified address is displayed and the cursor is positioned so you can modify the contents. Press the Return key to move to the next memory location. Press the Command key to execute the modification and return to SCI command mode. If the task is not unconditionally suspended, it is temporarily suspended while the command is executing.

Syntax:

```
[]MM
MODIFY MEMORY
      RUN ID:  integer expression ( )
     ADDRESS:  full expression list
```

The command displays the specified address and its value, followed by the next fifteen memory addresses and values. To modify a value, enter a new value in its place and press the Return key. Pressing the Return key moves the cursor to the next memory address. Pressing the Command key executes the command and returns to SCI command mode.

Example:

```
MODIFY MEMORY
       RUN ID:  >10
      ADDRESS:  >34B6

34B6:  >2FCF
34B8:  >0E9A
34BA:  >4449
34BC:  >4749
34BE:  >4F20
34C0:  >2020
34C2:  >0001
34C4:  >0060
34C6:  >494E
34C8:  >5055
34CA:  >5420
34CC:  >2020
34CE:  >34C6
34D0:  >0002
34D2:  >0013
34D4:  >454E
```

9.3.2 Pascal Debug Commands

The SCI includes seven commands for debugging Pascal tasks. These commands allow you to examine the stack area of a Pascal task and to set breakpoints independent of the code generated by the Pascal compiler. The commands are as follows:

- Assign Breakpoint — Pascal (ABP)
- Delete Breakpoint — Pascal (DBP)
- Delete and Proceed from Breakpoint — Pascal (DPBP)
- Proceed from Breakpoint — Pascal (PBP)
- List Breakpoints — Pascal (LBP)
- Show Pascal Stack (SPS)
- List Pascal Stack (LPS)

SCI writes error messages when it detects errors in the Pascal debug commands. Appendix B lists these messages.

9.3.2.1 Assign Breakpoint — Pascal (ABP). The ABP command assigns a breakpoint at entry to, return from, or both entry to and return from a Pascal routine. The syntax of the ABP command is as follows:

```
[] ABP
ASSIGN BREAKPOINT - PASCAL
            RUN ID: integer expression ( )
      ROUTINE NAME: character(s)
WHERE (ENTRY/RETURN/BOTH): {ENTRY/RETURN/BOTH}
```

The RUN ID is the system-assigned run-time ID of the Pascal task to be debugged. The ROUTINE NAME is the procedure, function, or program identifier of any Pascal or assembly language routine called by the Pascal task. Pascal routines must be compiled with the TRACEBACK option. Assembly language routines require the standard interface described in Section 11. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of one of these. ENTRY (or E) specifies a breakpoint at entry to the named routine; RETURN (or R) specifies a breakpoint at return from the routine; and BOTH (or B) specifies breakpoints at entry and return.

The first instruction in a routine compiled by the TIP compiler (or coded with the standard interface) is a Branch and Link (BL) instruction to a run-time routine for entry handling. When an ABP command that specifies either ENTRY or BOTH is executed, the command replaces the instruction following the BL instruction with a breakpoint (XOP 15,15). The last instruction in the routine is a B instruction to a run-time return handler. When an ABP command that specifies either RETURN or BOTH is executed, the command replaces the B instruction with a breakpoint (XOP 15,15).

Routines that are in an overlay can have breakpoints set only if that overlay is currently loaded in memory. When another overlay is loaded over it, any breakpoints in the first overlay are lost and will have to be reset the next time the overlay is loaded. For this reason, it is often necessary to set a breakpoint at the return from the overlay loader, OVLY$, to proceed until a particular overlay has been loaded, and then to set a breakpoint in it. Use the ABP command to set a breakpoint at the return from OVLY$, specifying OVLY$ as the routine name and R in response to the WHERE prompt.

Execution of a breakpoint suspends the task, placing it in state 6. When execution has been initiated by a DPBP command or a PBP command, a display appears on the VDT screen when the breakpoint is executed.

To continue execution following a breakpoint, enter one of the following:

- A PBP command
- A DPBP command
- A DBP command followed by an RT command

Example:

```
ASSIGN BREAKPOINT - PASCAL
            RUN ID:  >04
      ROUTINE NAME:  CCHAR
WHERE (ENTRY/RETURN/BOTH):  RETURN
```

The example command sets a breakpoint at the return from routine CCHAR of a task with the run-time ID of >04.

9.3.2.2. Delete Breakpoint — Pascal (DBP). The DBP command deletes a breakpoint at entry to, return from, or both entry to and return from a Pascal routine. The syntax of the DBP command is as follows:

```
          [] DBP
          DELETE BREAKPOINT - PASCAL
                    RUN ID:   integer expression ( )
               ROUTINE NAME:  character(s)
WHERE (ENTRY/RETURN/BOTH):    {ENTRY/RETURN/BOTH} (*)
```

The RUN ID is the system-assigned run-time ID of the Pascal task to be debugged. The NAME OF ROUTINE is either the procedure, function, or program identifier of a routine, or keyword ALL. When keyword ALL is entered, the WHERE prompt is optional and is ignored, and all breakpoints assigned to the task are deleted. Otherwise, the WHERE prompt is required, and must be either ENTRY, RETURN, or BOTH, or single-character abbreviations of one of these.

When a breakpoint has been set at the specified location, the DBP command replaces the breakpoint (XOP 15,15) with the original contents of the location. When the location does not contain a breakpoint, the command writes a warning message. If the task is suspended when the command is entered, the task does not resume execution. Enter an RT command to resume execution.

Example:

```
          DELETE BREAKPOINT - PASCAL
                    RUN ID:   >04
               ROUTINE NAME:  CCHAR
WHERE (ENTRY/RETURN/BOTH):    R
```

The example command deletes a breakpoint at the return of routine CCHAR in a task with the run-time ID of >04.

9.3.2.3 Delete and Proceed from Breakpoint — Pascal (DPBP). The DPBP command deletes the breakpoint that suspended the task, optionally assigns another breakpoint, and resumes execution of the task. The syntax of the DPBP command is as follows:

```
        [] DPBP
        DELETE AND PROCEED FROM BREAKPOINT - PASCAL
                    RUN ID: integer expression ( )
      DESTINATION ROUTINE: [character(s)] (*)
WHERE (ENTRY/RETURN/BOTH): [{ENTRY/RETURN/BOTH}] (*)
```

The RUN ID is the system-assigned run-time ID of the Pascal task being debugged. You may omit the other two parameters, in which case no breakpoint is assigned. The DESTINATION ROUTINE is the procedure or function identifier of a routine. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of one of these.

When the task has not been suspended by a breakpoint, the command assigns the specified breakpoint, if any, and writes a warning message. When the task is in the controlled mode, all activity at the terminal suspends until the task executes a breakpoint. Data similar to that displayed for an SPS command (paragraph 9.3.6) appears at a breakpoint (when the terminal is a VDT). When the task is not in the controlled mode, you must monitor task status with a Show Pascal Stack (SPS), Show Panel (SP), or Show Internal Registers (SIR) command (described in the *DNOS System Command Interpreter (SCI) Reference Manual*) to determine whether a breakpoint has executed.

Example:

```
 DELETE AND PROCEED FROM BREAKPOINT - PASCAL
                RUN ID:  >04
     DESTINATION ROUTINE:
WHERE (ENTRY/RETURN/BOTH):
```

The example above deletes the breakpoint that suspended task >04 and resumes execution of the task if it was suspended by a breakpoint. If the task was not suspended by a breakpoint, the command displays a warning message.

```
 DELETE AND PROCEED FROM BREAKPOINT - PASCAL
                RUN ID:  >04
     DESTINATION ROUTINE:  CINT
WHERE (ENTRY/RETURN/BOTH):  BOTH
```

The example above deletes the breakpoint, assigns breakpoints at the entry and return of routine CINT of task >04, and resumes execution of the task. If task >04 was not suspended by a breakpoint, the command assigns the breakpoint and then displays a warning message.

9.3.2.4. Proceed from Breakpoint — Pascal (PBP). The PBP command optionally assigns a breakpoint and resumes execution of the task. The syntax of the PBP command is as follows:

```
[] PBP
PROCEED FROM BREAKPOINT - PASCAL
                RUN ID:  integer expression ( )
     DESTINATION ROUTINE:  [character(s)] (*)
WHERE (ENTRY/RETURN/BOTH):  [{ENTRY/RETURN/BOTH}] (*)
```

The prompts are the same as for the DPBP command.

When the task is in the controlled mode, all activity at the terminal suspends until the task executes a breakpoint. Data similar to that displayed for an SPS command (paragraph 9.3.6) appears at a breakpoint (when the terminal is a VDT). When the task is not in the controlled mode, you must monitor task status with an SPS, SP, or SIR command to determine whether a breakpoint has executed.

Example:

```
PROCEED FROM BREAKPOINT - PASCAL
                RUN ID:  >04
     DESTINATION ROUTINE:
WHERE (ENTRY/RETURN/BOTH):
```

The example above resumes execution of task >04 without altering the number of active breakpoints.

```
  PROCEED FROM BREAKPOINT - PASCAL
                  RUN ID:  >04
      DESTINATION ROUTINE:  CINT
WHERE (ENTRY/RETURN/BOTH):  E
```

The example above assigns a breakpoint at the entry to routine CINT of task >04 and resumes execution of the task.

9.3.2.5 List Breakpoints — Pascal (LBP). The LBP command lists the breakpoints assigned for a task. The syntax of the LBP command is as follows:

```
[] LBP
LIST BREAKPOINTS - PASCAL
          RUN ID:  integer expression ( )
```

The RUN ID is the system-assigned run-time ID of the Pascal task for which breakpoints are to be listed.

Each entry of the listing describes the Pascal breakpoint(s) for a routine of the task. The format for each entry of the listing is as follows:

```
NNNNNNNN L EEEE RRRR
```

The routine name is represented by NNNNNNNN. The letter L represents E (entry breakpoint), R (return breakpoint), or B (both entry and return breakpoints). Field EEEE contains the hexadecimal address within the task of the entry breakpoint when L is either E or B. Field RRRR contains the hexadecimal address within the task of the return breakpoint when L is either R or B. A maximum of three entries can be displayed on a line of the listing.

Example:

```
LIST BREAKPOINTS - PASCAL
          RUN ID:  >04
```

The example command specifies listing all Pascal breakpoints assigned for task >04; the output is similar to the following:

```
DIGIO   B 34F8 35CC    CINT   B   3472 34B6    CCHAR   B   3472 34B6
```

9.3.2.6 Show Pascal Stack (SPS). The SPS command displays data from the stack structure of the currently executing process, as well as other data pertaining to the specified task. The syntax of the SPS command is as follows:

```
[] SPS
SHOW PASCAL STACK
           RUN ID:  integer expression ( )
```

The RUN ID is the system-assigned run-time ID of the Pascal task for which the stack is to be displayed.

Example:

```
SHOW PASCAL STACK
          RUN ID:  >2B
```

The example command specifies displaying the stack structure for task >2B.

The displayed data includes the contents of the internal registers, the breakpoints, and the stack structure of all processes beginning with the current process. Figure 9-5 shows an example of the display.

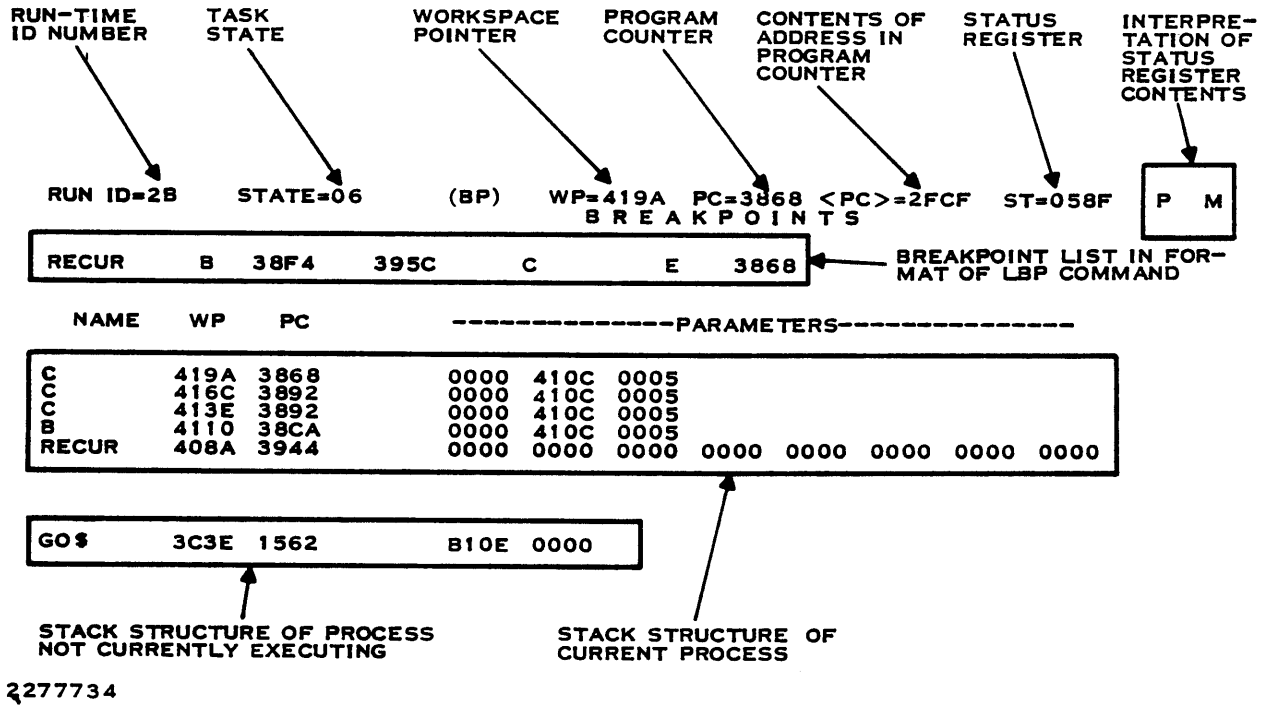


Figure 9-5. Show Pascal Stack (SPS) Display

The first line of the display shows the run-time ID, the task state, the contents of the WP and the PC, the contents of the address in the PC, the contents of the ST, and the interpretation of the ST contents. In the example, (BP) follows the task state to indicate that the task is in unconditional suspension due to the execution of a breakpoint. The address in the PC is the address of the next instruction to be executed when execution resumes. The contents of the address is the next machine instruction to be executed. The letters P and M indicate that the parity and map file bits of the ST have been set to one. The letters that appear in this field can include the following:

- L = logical greater than
- A = arithmetic greater than

- E = equal
- C = carry
- O = overflow
- P = parity
- X = XOP in progress
- S = privileged mode
- M = map file

The next portion of the display lists the assigned breakpoints (in the format defined for the LBP command). Three breakpoints are listed in the example: two in the program module RECUR and one in routine C. The contents of the PC and the address of the breakpoint at the entry to routine C are the same; the breakpoint in routine C is the breakpoint that produces this display.

The remainder of the display consists of the stack structures of the processes involved in executing the task. For more information on processes, refer to the description of the abnormal termination dump earlier in this section. The stack structure of the currently executing process appears first. Each line of the stack structure portion of the display corresponds to the stack frame of an active routine, listed in the reverse order in which they were activated. In Figure 9-5, program module RECUR called routine B, which called recursive routine C. The display shows the stack structure at entry of the second time routine C called itself. The first line corresponds to the second recursive call of routine C, the second line to the first recursive call, the third line to the call of routine C by routine B, the fourth line to the call of routine B, and the last line to the program module RECUR.

Each line of the stack structure lists the contents of WP and PC and the parameters for the routine. The address in the WP is also the address of the bottom (lowest address) of the stack frame for the routine. The address in the PC is the address at which execution of the routine resumes when control returns to the routine. The parameters listing shows the contents of up to eight words of the stack frame, beginning at address >28 relative to the bottom of the stack frame. When the stack frame contains fewer than >30 words, only the words from relative address >28 to the top of the frame are displayed. When a routine has eight or more parameters, eight (or the first eight) words of the parameters are displayed. When the routine has fewer than eight parameters, all of the parameters are displayed, followed by local data (if any is available). The word that corresponds to a value parameter contains the value of the parameter; the word that corresponds to a reference parameter contains the address of the parameter. In the example, routine C has three parameters and no local data. Parameter 1 is a value parameter with a value of zero, parameter 2 is a reference parameter at location >410C, and parameter 3 is a value parameter with a value of 5.

A blank line separates the stack structure for the currently executing process from that of the next process. The information provided is not as useful for debugging purposes since you are not concerned with debugging the routines in these processes. In the example, the stack structure of the nonexecuting process contains a single stack frame. When there is more than one nonexecuting process, more than one additional stack structure is displayed; a blank line separates each stack structure display from that of the next stack structure.

9.3.2.7. List Pascal Stack (LPS) The LPS command lists a specified portion of the stack frame for a specified routine. The listing is written to a specified device. The syntax of the LPS command is as follows:

```
[] LPS
LIST PASCAL STACK
            RUN ID:  integer expression ( )
      ROUTINE NAME:  [character(s)] (*)
   STARTING OFFSET:  integer expression (0)
   NUMBER OF BYTES:  [integer expression] (*)
LISTING ACCESS NAME:  [pathname@] (stationname) (*)
```

The RUN ID is the system-assigned run-time ID of the Pascal task for which the stack is to be listed. The ROUTINE NAME is optional. If you enter a routine name, the stack frames for the named routines in all processes are listed. If you do not enter a routine name, all stack frames of all processes are listed. The STARTING OFFSET is the relative address within each stack frame of the first word to be listed. The NUMBER OF BYTES is optional. If you enter a number of bytes, the specified number of bytes of each stack frame is listed. If you do not enter a number or if you enter a number that is greater than the number of bytes the stack frame contains, the command lists the contents from the starting offset to the top of the frame. The LISTING ACCESS NAME is the pathname of a file or device to which the listing is to be written. This parameter is optional; when it is omitted, the list is written to the terminal local file (TLF) of the terminal and is then displayed.

If the task specified in the command is not unconditionally suspended, the task is suspended while the listing is written. The listing consists of a heading and a variable number of lines for each stack frame. The format of the heading is as follows:

LIST STACK FRAME OF NNNNNNNN. FRAME BEGINS AT AAAA.

NNNNNNNN represents the name of the routine (this field will be blank if the routine was compiled without the TRACEBACK option). AAAA represents the address of the bottom of the stack frame.

The format of the lines that follow the heading is as follows:

RRRR DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD CCCCCCCCCCCCCCCC

RRRR represents the relative address within the stack frame of the first word listed. Each DDDD represents the contents of a word of the stack frame, listed as four hexadecimal digits. Each C represents the contents of a byte, represented either as an ASCII character or as a period (.) if the ASCII character is unprintable.

The number of lines listed for each stack frame depends on the number of bytes to be listed. One line is listed when 16 or fewer bytes are to be listed. When more than 16 bytes are to be listed, they are listed 16 bytes per line with any remaining bytes on an additional line.

The stack frames for the currently executing process are listed first, followed by those for processes that are not executing. When the specified routine is executing or suspended or when all routines are being listed, the first stack frame listed is that of the current routine. Otherwise, the stack frame that corresponds to the most recent call of the routine is listed first. Additional stack frames are listed in the reverse of the order in which they were called. A blank line separates the listings of stack frames of one process from those of another process.

When a listing of stack frames for a task or routine that is not active is requested, the command writes an error message.

Example:

```
LIST PASCAL STACK
            RUN ID:  >2B
      ROUTINE NAME:
    STARTING OFFSET:  >28
    NUMBER OF BYTES:
LISTING ACCESS NAME:  MYVOL.PASCAL.STACKS.TASK1
```

The example above lists on one file the stack frames of all routines of task >2B. The entire contents of the stack frame starting with byte >28 relative to the bottom of the stack are listed. The access name of the file is MYVOL.PASCAL.STACKS.TASK1.

```
LIST PASCAL STACK
            RUN ID:  >2B
      ROUTINE NAME:  B
    STARTING OFFSET:  >28
    NUMBER OF BYTES:  10
LISTING ACCESS NAME:
```

The example above lists on the terminal screen the stack frames for routine B of task >2B. Ten bytes are requested, starting at byte >28 of each stack frame. As many as 16 bytes can be listed per stack frame. The requested number of bytes requires one line. If 16 or more bytes are included from the starting byte to the top of the stack, 16 bytes are listed, filling the line. Otherwise, the bytes from the starting byte to the top of the stack are listed.

9.3.3 Using Debug Commands

The example procedure in this paragraph uses the following commands to debug the example program DIGIO:

```
XPT, XD, ABP, RT, SPS, LM, MM, AND PBP
```

The program DIGIO accepts five characters as input and yields a predictable output value. The input BBBBB yields the output 25, and the input 99999 yields the output –/0,(. In the example, the file INPUT contains BBBBB. The debugging example demonstrates executing DIGIO in controlled mode, changing the input values to 99999, and then resuming program execution to check the result of the modification. The steps are as follows:

1. Execute DIGIO by using the XPT command.

```
[] XPT
EXECUTE TI PASCAL TASK
        PROGRAM FILE:  D.EXPROG
      TASK NAME OR ID:  DIGIO
               INPUT:  D.DATA
              OUTPUT:
            MESSAGES:
         MODE(F,B,D):  D
              MEMORY:
```

The display above specifies the program file and task for the example, specifies the input file D.DATA (which contains the value BBBBB), accepts default output and message files, specifies debug mode, and accepts the default memory allocations. The display that appears when the XPT command executes indicates the run ID of the task, along with other information.

2. Enter the XD command to place the task in controlled mode, as follows:

```
[] XD
INITIATE DEBUG MODE
                    RUN ID:  >8A
SYMBOL TABLE OBJECT FILE:
     990/12 OBJECT CODE?:
```

The Run ID value automatically appears in response to the prompt RUN ID; accept the defaults for the remaining prompts. The XD command initially displays the program's workspace registers and a portion of memory beginning with the current memory location. It displays the stack as in the SPS command after executing a breakpoint.

3. Assign breakpoints by using the ABP command. In this example, a breakpoint is set at the entry of routine CCHAR, before CCHAR performs the initial data manipulation.

```
[] ABP
   ASSIGN BREAKPOINT - PASCAL
                      RUN ID:  >8A
                ROUTINE NAME:  CCHAR
  WHERE (ENTRY/RETURN/BOTH):  E
```

4. Enter the RT command to allow the program to execute to the breakpoint.

```
[] RT
   RESUME TASK
               RUN ID:  >8A
```

The task executes to the breakpoint at the entry of routine CCHAR. Then, the task suspends itself and displays the workspace registers, breakpoint(s), and a portion of memory beginning at the current location.

The following is an example:

```
RUN ID=8A STATE=06 (BP) WP=50BA  PC=3400 <PC>=2FCF  ST=C18F  LA   M
                        WORKSPACEREGISTERS
50BA 0050  FFFF  0013  4FB5  34E7  0000  508A  0000  .P ..  ..0.4...P...
50CA 50A0  50BA  50F4  3400  54AC  502A  35AE  C18F  P. P.  P.4.T.P*5...
                           BREAKPOINTS
   3400
                              MEMORY
3400 2FCF  0034  04D5  0206  0001  CA69  0036  0038  /. .4  .......i.6.8
3410 8A46  0038  1522  C146  0A15  0225  0026  A149  .F .8  .  .F..%.&.I
3420 8315  33F2  1A18  C146  0A15  0225  0226  A149  .. 3.  ." .F..%.&.I
3430 8815  33F4  1B10  C169  0034  0203  000A  38D5  .. 3.  ...i.4....8.
      "
      "
      "
```

5. Execute the SPS command to display the stack structure for the process that is currently executing, as in the following example:

```
[] SPS
   SHOW PASCAL STACK
              RUN ID:  >8A


RUN ID=8A  STATE=06 (BP)  WP=50BA  PC=3400  <PC>=2FCF  ST=C18F  LA   M
                                 BREAKPOINTS
 CCHAR  E    3400

  NAME WP   PC          --------PARAMETERS-------
CCHAR 50BA 3400     0042 0042 0042 0042 0042 0000 50B8 0005
DIGIO 502A 35AE     0000 0000 0000 0000 0000 0000 0000 0000

GO$   422A 13B0     54AC 0000
```

The map of identifiers on the compiler source listing for DIGIO (Figure 5-3) shows that the variables for routine CCHAR are BUFF, consisting of 12 bytes at displacement >28, NUM, with 2 bytes at displacement >34, and I, with 2 bytes at displacement >36. The stack display for CCHAR shows the value >0042 in the first five words in the stack, corresponding to the ASCII character B in the first five positions in the input buffer BUFF. The sixth word contains zeros. Since NUM is passed by reference, the value in the seventh word of the parameters display is an address, >50B8. The value of I, shown in the eighth word, is 5, the number of characters input.

6. You can show the value of NUM by using the LM command. Specify the address shown in the seventh word of the SPS display above, as follows:

```
[] LM
   LIST MEMORY
                    RUN ID:  >8A
          STARTING ADDRESS:  >50B8
            NUMBER OF BYTES:
        LISTING ACCESS NAME:
```

The memory listing is as follows:

```
50B8  0000  0050  FFFF  0013  4FB5  34E7  0000  508A  ...P....0.  4...P.
```

7. Use the MM command to change the values in BUFF. Knowing that BUFF begins at displacement >28 allows you to enter the starting address for the MM command as an expression: #WP$ + >28. The expression resolves to the beginning address of the workspace for CCHAR plus >28 bytes.

```
[] MM
   MODIFY MEMORY
            RUN ID:  >8A
           ADDRESS:  #WP+>28
```

The MM display is as follows:

```
50E2:  >0042
50E4:  >0042
50E6:  >0042
50E8:  >0042
50EA:  >0042
50EC:  >0000
50EE:  >50B8
50F0:  >0005
50F2:  >0004
50F4:  >5146
50F6:  >50E4
50F8:  >5112
50FA:  >4078
50FC:  >54AC
50FE:  >50BA
5100:  >3F64
```

For this example, each value >0042 is changed to >0039 by entering the new value over the old one and pressing the Return key. (If you do not press the Return key, the modification does not occur.) After changing the values, the display appears as follows:

```
50E2:  >0039
50E4:  >0039
50E6:  >0039
50E8:  >0039
50EA:  >0039
50EC:  >0000
50EE:  >50B8
50F0:  >0005
50F2:  >0004
50F4:  >5146
50F6:  >50E4
50F8:  >5112
50FA:  >4078
50FC:  >54AC
50FE:  >50BA
5100:  >3F64
```

When all changes are made, pressing the Command key executes the modifications and displays the internal registers again.

8. You can enter the SPS command again to verify the modifications, as follows:

```
[] SPS
   SHOW PASCAL STACK
              RUN ID: >8A


RUN ID=8A STATE=06 (BP) WP=50BA  PC=3400  <PC>=2FCF  ST=C18F  LA    M
                          B R E A K P O I N T S
  CCHAR  E  3400

   NAME WP PC      ------------PARAMETERS----------
CCHAR 50BA 3400   0039 0039 0039 0039 0039 0000 50B8 0005
DIGIO 502A 35AE   0000 0000 0000 0000 0000 0000 0000 0000

GO$   422A 13B0   54AC 0000
```

9. If no further debugging is desired at this point, you can use the DBP command to delete the current breakpoint. Next, execute the QD command to take the task out of controlled mode. Finally, use the RT command to allow the task to complete execution. Alternately, the PBP command accomplishes the same result, as in this example. The PBP command allows you to set another breakpoint (destination address) if desired. The PBP command is as follows:

```
[] PBP
   PROCEED FROM BREAKPOINT- PASCAL
                        RUN ID:  >8A
         DESTINATION ADDRESS:
    WHERE (ENTRY/RETURN/BOTH):
```

10. Check the output of the task by using the SF command, as follows:

```
[] SF
   SHOW FILE
        FILE PATHNAME: .OUTPUT16
```

The contents of .OUTPUT16 are as follows:

```
ENTER 1 TO 5 DIGITS
-/0,(
```

The modifications performed during debugging are made to the memory image of the program, not to the image on the program file. If debugging shows that the program code must be changed, you should revise the source code, recompile it, and relink.

9.4 RUN-TIME LIBRARY ROUTINES

When debugging a program, you might find it helpful to know the functions that certain run-time routines perform. For example, a memory dump might show that an error occurred in a library routine, and knowing the routine's function should help you understand the dump. Also, you might want to assign breakpoints (ABP command) on run-time routines. Table 9-1 lists run-time routines whose names might appear on a dump. These are also routines on which you might assign a breakpoint. The list does not include the run-time library routines from Section 10 or those that correspond directly to predefined functions and procedures listed in the *TI Pascal Reference Manual*, except for the addition of a $ to the routine name (for example, routine SQRT$ implements the SQRT function).

Of particular interest is routine ABEND$, the abnormal termination routine. Setting a breakpoint at the entry to ABEND$ suspends execution at that point when an error occurs (except for task errors and a few other special cases). You can then use the debug commands SPS, LPS, and LM to diagnose the problem. Register 0 in ABEND$ is the error code, as listed in Appendix B.

In Table 9-1, the types of routine parameters are indicated by letters with the following meanings and sizes:

Type	Meaning	Size
A	Memory address passed by value	1 word
B	Buffer address	1 word
D	Rounding flag — 1 = round, 0 = do not round	1 word
F	File descriptor address	1 word
G	Eight-character name passed by value	4 words
I	INTEGER value	1 word
J	LONGINT value	2 words
L	Length of variable — number of bytes	1 word
N	Source line number	1 word
P	Precision — number of bits or digits	1 word
Q	Scale Factor — number of bits or digits	1 word
R	Record number (LONGINT value)	2 words
S	Address of SVC block	1 word
T	Address of pointer variable	1 word
U	Address of INTEGER variable with column number	1 word
V	Address of variable where result will be returned	1 word
W	Field width value	1 word
X	Address of input variable	1 word
Z	Address of status variable	1 word

Table 9-1 Miscellaneous Run-Time Routines

Routine	Purpose
ABEND$(I)	Abnormal termination, I = error code
ASSER$(N)	ASSERT failure
BOSF$(I,S,V)	SKIPFILES
CLOSE$(F)	Close file
CLS$(F)	Close and deallocate file
CLS$FILE(S)	Close file
CSLAB$(N,I)	CASE alternative error
CVTD$F(X,P,Q,V,P,Q)	Convert DECIMAL to FIXED
CVTD$L(X,P,Q,V)	Convert DECIMAL to LONGINT
CVTD$R(X,P,Q,V)	Convert DECIMAL to REAL
CVTE$F(X,V,P,Q)	Convert double REAL to FIXED
CVTF$D(X,P,Q,V,P,Q)	Convert FIXED to DECIMAL
CVTF$E(X,P,Q,V)	Convert FIXED to double REAL
CVTF$R(X,P,Q,V)	Convert FIXED to REAL
CVTL$D(X,V,P,Q,D)	Convert LONGINT to DECIMAL
CVTR$D(X,V,P,Q,D)	Convert REAL to DECIMAL
CVTR$F(X,V,P,Q)	Convert REAL to FIXED
DEB$T(B,W,U,Z,L,V)	DECODE BOOLEAN
DEC$T(B,W,U,Z,L,V)	DECODE CHAR
DED$T(B,W,P,Q,U,Z,L,V)	DECODE DECIMAL
DEF$T(B,W,P,Q,U,Z,L,V)	DECODE FIXED
DEI$T(B,W,U,Z,L,V)	DECODE INTEGER
DEL$T(B,W,V,Z,L,V)	DECODE LONGINT
DER$T(B,W,P,U,Z,L,V)	DECODE REAL
DES$T(B,W,L,U,Z,L,V)	DECODE string
DEX$T(B,W,L,U,Z,L,V)	DECODE hexadecimal
ENB$T(B,W,U,Z,L,I)	ENCODE BOOLEAN
ENC$T(B,W,U,Z,L,I)	ENCODE CHAR
END$T(B,W,I,P,Q,U,Z,L,X)	ENCODE DECIMAL
ENF$T(B,W,I,P,Q,V,Z,L,X)	ENCODE FIXED
ENI$T(B,W,U,Z,L,I)	ENCODE INTEGER
ENL$T(B,W,V,Z,L,J)	ENCODE LONGINT
ENR$T(B,W,I,P,U,Z,L,X)	ENCODE REAL
ENS$T(B,W,I,U,Z,L,X)	ENCODE string
ENX$T(B,W,I,V,Z,L,X)	ENCODE hexadecimal
EXTND$(F)	EXTEND
FL$ERR(I,I,I)	Floating-point error
FL$INIT(F,G,I,L)	Initialize file descriptor
FREE$(T)	DISPOSE
GET$RCOR(S,B,L,Z)	Read record from sequential or text file
GET$RLRE(S,B,R,L,Z)	Read record from random file
GO$	Initiate/terminate process
HALT$	HALT
INIT$(T)	Create and initialize user process
IO$ERR(F,I)	Report I/O error
MESAG$DX(X,L)	MESSAGE — SCI version
MESAG$EX(X,L)	MESSAGE — EXTMSG version
MESAG$LU(X,L)	MESSAGE — LUNOBJ version
MESAG$MI(X,L)	MESSAGE — MINOBJ version
NEW$(T,L)	NEW

Table 9-1 Miscellaneous Run-Time Routines (Continued)

Routine	Purpose
OPEN$(F,I)	Open file
OPN$FILE(S,I,V,Z)	Open file
OV$FLO(N)	Report overflow error
P$TERM	Terminate execution
PREC$$(N)	Precision-check failure
PRT$MEM(A,A)	Dump block of memory
PUT$RCOR(S,B,L,Z)	Write logical record
PUT$RLRE(S,B,R,L,Z)	Write record to random file
RDB$T(F,W,V)	READ BOOLEAN from text file
RDC$T(F,W,V)	READ CHAR from text file
RDD$T(F,W,P,Q,V)	READ DECIMAL from text file
RDF$T(F,W,P,Q,V)	READ FIXED from text file
RDI$T(F,W,V)	READ INTEGER from text file
RDL$T(F,W,V)	READ LONGINT from text file
RDLN$(F)	READLN
RDR$T(F,W,P,V)	READ REAL from text file
RDS$T(F,W,L,V)	READ string from text file
RDX$T(F,W,L,V)	READ hexadecimal from text file
READ$(F,B)	READ sequential file
READ$REL(F,R,B)	READ random file
RESET$(F)	RESET
REST$R(F)	RESET random file
REST$S(F)	RESET sequential file
REST$T(F)	RESET text file
RETRN$(G)	ESCAPE from routine
REWND$(F)	Rewind file
REWRT$(F)	REWRITE
SCB$FREE(T)	Deallocate SVC block
SCB$INIT(T,I)	Allocate and initialize SVC block
S$NAME(S,G)	SETNAME
SVC$(S)	Supervisor Call
TERM$(A)	Write termination messages
TX$ERR(F,I)	Report text file error
WRB$T(F,W,I)	WRITE BOOLEAN to text file
WRC$T(F,W,I)	WRITE CHAR to text file
WRD$T(F,W,I,P,Q,X)	WRITE DECIMAL to text file
WRF$T(F,W,I,P,Q,X)	WRITE FIXED to text file
WRI$T(F,W,I)	WRITE INTEGER to text file
WRITE$(F,B)	WRITE to sequential file
WRL$T(F,W,J)	WRITE LONGINT to text file
WRLN$(F)	WRITELN
WRR$T(F,W,I,P,X)	WRITE REAL to text file
WRS$T(F,W,L,X)	WRITE string to text file
WRT$REL(F,R,B)	WRITE to random file
WRX$T(F,W,L,X)	WRITE hexadecimal to text file

10

Run-Time Library Routines

10.1 GENERAL

The routines described in this section are not part of TIP but are computer dependent. This section describes the routines provided for direct communication register unit (CRU) I/O and for interface with the 990 operating systems.

10.2 DIRECT CRU I/O ROUTINES

The direct CRU I/O routines enable you to perform I/O operations with CRU devices. You can use these routines with CRU devices that are not supported by the operating system or by stand-alone TIP tasks. The CRU routines correspond to the CRU instructions of assembly language, as follows:

- Procedure $LDCR corresponds to a Load CRU (LDCR) instruction
- Procedure $SBO corresponds to a Set CRU Bit to Logic One (SBO) instruction
- Procedure $SBZ corresponds to a Set CRU Bit to Logic Zero (SBZ) instruction
- Procedure $STCR corresponds to a Store CRU (STCR) instruction
- Function $TB corresponds to a Test Bit (TB) instruction

NOTE

The CRU base address for each of these routines is the CRU (hardware) address. The hardware address is determined by dividing the value (specified during system generation or written on the chassis) by 2. Each CRU I/O routine multiplies the value by 2 and places the product in workspace register 12 for the operation.

10.2.1 Procedure $LDCR

A routine that calls procedure $LDCR to output data to a CRU device must declare $LDCR among the declarations for the routine, as follows:

```
PROCEDURE $LDCR(BASE,WIDTH,VALUE: INTEGER); EXTERNAL;
```

Assuming that B and V have been defined as integer variables, the following example shows a call to $LDCR:

```
B:=#200;                    (*set base to #200*)
V:=#41;                     (*character is #41 (A)*)
$LDCR(B,8,V);               (*output character (8 bits)*)
```

10.2.2 Procedure $SBO

A routine that calls procedure $SBO to set a CRU bit to one must declare $SBO among the declarations for the routine, as follows:

```
PROCEDURE $SBO(BASE: INTEGER); EXTERNAL;
```

Assuming that ADDR has been declared as an integer variable, the following example shows a call to $SBO:

```
ADDR:=#400;                 (*set base to #400*)
ADDR:=ADDR + 4;             (*add displacement*)
$SBO(ADDR);                 (*set bit to one*)
```

10.2.3 Procedure $SBZ

A routine that calls procedure $SBZ to set a CRU bit to zero must declare $SBZ among the declarations for the routine, as follows:

```
PROCEDURE $SBZ(BASE: INTEGER); EXTERNAL;
```

Assuming that BITOUT has been declared as an integer variable, the following example shows a call to $SBZ:

```
BITOUT:=#400;               (*set base to #400*)
BITOUT:=BITOUT + 2;         (*add displacement*)
$SBZ(BITOUT);               (*set bit to zero*)
```

10.2.4 Procedure $STCR

A routine that calls procedure $STCR to input data from a CRU device must declare $STCR among the declarations for the routine, as follows:

```
PROCEDURE $STCR(BASE,WIDTH: INTEGER; VAR VALUE: INTEGER);
          EXTERNAL;
```

The following is an example of the use of the $STCR procedure:

```
CONST BS1 =#200;
  SIZE = 8;
VAR INCHR: INTEGER;
 . . .
$STCR(BS1,SIZE,INCHR);      (*read character*)
```

10.2.5 Function $TB

A routine that calls function $TB to input a bit from a CRU device must declare $TB among the declarations for the routine, as follows:

```
FUNCTION $TB(BASE: INTEGER):BOOLEAN; EXTERNAL;
```

Assuming that ADD has been declared as type INTEGER and BUSY has been declared as type BOOLEAN, the following is an example of the use of function $TB:

```
ADD: = #200;              (*set base to #200*)
ADD: = ADD + 8;           (*add displacement*)
BUSY: = $TB(ADD);         (*set BUSY to value of CRU input*)
```

10.3 SYSTEM COMMAND INTERPRETER (SCI) INTERFACE ROUTINES

The following routines allow you to access SCI to find and set synonym values and to display messages.

10.3.1 Procedure FIND$SYN

Procedure FIND$SYN obtains the value of an SCI synonym. The procedure is a TIP interface to system routine S$MAPS. Place the maximum length of the value string (as limited by the size of the array into which it is placed) into element 0 of the value string; then, call the procedure.

The declaration of type SCI_STRING is as follows:

```
TYPE SCI_STRING = PACKED ARRAY [0..N] OF CHAR;
```

where:

N represents an integer constant; do not use "?" for the upper bound.

The declaration for procedure FIND$SYN is as follows:

```
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE: SCI_STRING); EXTERNAL;
```

This procedure has two parameters: a string that contains the synonym of the value to be obtained, and a string into which FIND$SYN places the value. FIND$SYN accesses the terminal communications area (TCA), searches the synonym table for the specified synonym, and returns the value. If the synonym is found, the number of characters in the value is placed in element 0 of the value string, and the characters of the value are placed in elements 1 through N of the value string. If the synonym is not found, zero is placed in element 0 of the value string. For tasks linked for execution in a non-SCI environment, FIND$SYN always returns a zero in element 0 of the value string, indicating that the synonym was not found.

An example of the use of procedure FIND$SYN is as follows:

```
CONST SL = 80;                                          (*maximum length of SCI_STRING*)
      N = 3;                                            (*size of synonym*)
TYPE SCI_STRING = PACKED ARRAY [0..SL] OF CHAR;         (*required type*)
VAR SYN:SCI_STRING;                                     (*for synonym parameter*)
    ACCESS_NAME:SCI_STRING:                             (*for value parameter*)
    SYNA:PACKED ARRAY[1..N] OF CHAR;
            .
            .
            .
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE:
SCI_STRING);EXTERNAL;                                   (*external declaration*)
            .
            .
            .
BEGIN
            .
            .
            .
SYNA:='ONE';
FOR K:=1 TO N DO
  SYN[K]:=SYNA[K];                                      (*set up synonym name*)
SYN[0]:=CHR(N);                                         (*set up length of synonym*)
ACCESS_NAME[0]:=CHR(SL);
                                                        (*set maximum length*)
FIND$SYN(SYN,ACCESS_NAME);
                                                        (*call FIND$SYN*)
IF ORD(ACCESS_NAME[0]) = 0 THEN                         (*terminate abnormally if no synonym*)
```

10.3.2 Procedure STORE$SYN

Procedure STORE$SYN assigns a value to an SCI synonym. The procedure is a TIP interface to system routines S$GTCA, S$SETS, and S$PTCA. To use STORE$SYN, declare type SCI_STRING and declare the procedure externally.

The declaration for procedure STORE$SYN is as follows:

```
PROCEDURE STORE$SYN (VAR SYNONYM, VALUE:SCI_STRING); EXTERNAL;
```

This procedure has two parameters: a string that contains the synonym and a string that contains the value. STORE$SYN reads the TCA from disk, adds the synonym and value to the synonym table, and writes the updated data to the disk. This procedure applies only to programs executing under SCI.

If STORE$SYN is being used to change the access name for a Pascal file, then a call to SETNAME or SETMEMBER should follow the call to STORE$SYN in order to force reevaluation of the file's access name.

An example of the use of procedure STORE$SYN is as follows:

```
CONST SL = 80;                                          (*maximum length of SCI_STRING*)
      N = 3;                                            (*size of synonym*)
      NP = 11;                                          (*size of pathname*)
TYPE SCI_STRING = PACKED ARRAY [0..SL] OF CHAR;         (*required type*)
VAR SYN:SCI_STRING;                                     (*for synonym parameter*)
    ACCESS_NAME:SCI_STRING;                             (*for value parameter*)
    SYNA:PACKED ARRAY[1..N] OF CHAR;
    ACNM:PACKED ARRAY[1..NP] OF CHAR;
          .
          .
          .
PROCEDURE STORE$SYN(VAR SYNONYM, VALUE:SCI_STRING);EXTERNAL;
                                                        (*external declaration*)
          .
          .
          .
BEGIN
          .
          .
          .
  SYNA:='ONE';
  ACNM:='.INPUT. TEST';
  FOR K:=1 TO N DO
    SYN[K]:=SYNA[K];                                    (*set up synonym name*)
  SYN[0]:=CHR(N);                                       (*set length of synonym*)
  FOR K:=1 TO NP DO
    ACCESS_NAME[K]:=ACNM[K];
                                                        (*set up value*)
  ACCESS_NAME[0]:=CHR(NP);
                                                        (*set length of value*)
  STORE$SYN(SYN,ACCESS_NAME);
                                                        (*call STORE$SYN*)
```

10.3.3 Procedure P$UC

The system-defined synonym $$CC stores system and user condition codes. A task can call P$UC to set $$CC to one of the following values:

Value	Meaning
0000	Procedure called with a parameter of 0 or not called, and task terminated normally
4000	Procedure called with a parameter of 1, and task terminated normally
6000	Procedure called with a parameter of 2, and task terminated normally
8000	Procedure called with a parameter of 3, and task terminated normally
A000	Procedure called with a parameter of 4, and task terminated normally
C000	Task terminated abnormally or procedure HALT called

Test synonym $$CC to detect whether the task terminated normally or whether procedure P$UC was called by something other than a parameter of zero during task execution. Test or store the synonym before executing any of the operating system utilities that set the synonym. (Specifically, the List Synonym (LS) SCI command always lists the value of $$CC as zero because the command utility sets $$CC to zero.) You can use the .SYN SCI primitive to set another synonym to the value of @$$CC, as follows

```
.SYN S = @$$CC
```

Declare procedure P$UC externally, as follows:

```
PROCEDURE P$UC (CODE: INTEGER);EXTERNAL;
```

The parameter is a value in the range of 0 through 4 and determines the value to which $$CC is set, as previously described.

10.3.4 Procedure R$TERM

R$TERM is an interface to the DNOS SCI routine S$TERM, which terminates task execution and optionally specifies a message to be displayed. R$TERM closes any open TIP files before calling S$TERM. Declare R$TERM as follows:

```
TYPE BYTE = 0..#FF;
     STRING_REC = PACKED RECORD
                  MAXLEN: BYTE;
                  LENGTH: BYTE;
                  CH: PACKED ARRAY [1..255] OF CHAR;
                  END;
   * RSTRING = @STRING_REC;

PROCEDURE R$TERM(CC: INTEGER;
    VT: RSTRING; ES: INTEGER;
    MN: INTEGER); EXTERNAL;
```

The parameters are as follows:

Parameter	Description
CC	The value to which the completion code synonym $$CC will be set (This overrides a value set by procedure P$UC.)
VT	Pointer to variable text for termination message; NIL if no message text.
ES	Error souce flags, usually a message file number.
MN	Message number; zero (0) if no message.

Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for a description of how S$TERM uses the VT, ES, and MN parameters to build a message.

R$TERM can also be used in a non-SCI environment, in which case it does not call S$TERM but simulates its action using message definition modules linked with the task. Refer to the *Guide to the TI Pascal Run-time Support System* for further details.

10.3.5 Procedure P$PARM

Procedure P$PARM is an interface to the SCI routine S$PARM. It is used to obtain the value of a parameter of the .BID or .QBID SCI command that initiated the task. The declaration is as follows:

```
PROCEDURE P$PARM (NUMBER : INTEGER;
                  VAR STRING : PACKED ARRAY[1..?] OF CHAR;
                  VAR ERROR : INTEGER); EXTERNAL;
```

The first parameter is the number of the desired .BID or .QBID parameter. (The stack and heap sizes are parameters 1 and 2. So user parameters begin with number 3.) The second parameter is a variable where the parameter value is returned. The first array element is set to a binary value indicating the number of characters that follow. The third parameter is a variable that is set to zero if the operation succeeds, or to an error code returned by S$PARM.

10.4 KEY INDEXED FILE (KIF) HANDLING

A single procedure, KEY$FILE, enables a TIP program to perform a variety of operations on key indexed files. Each call to KEY$FILE includes a command code specifying the operation to be performed. The following paragraphs describe the use of KEY$FILE.

10.4.1 Procedure KEY$FILE

Procedure KEY$FILE has a single entry point for all calls. Since TIP routines must have the same number of parameters in each call, all calls to KEY$FILE must specify the following four parameters:

- KIF_COMMAND — Specifies what operation is to be done
- KIF_KEY_NAME — Indicates the value of the key to be processed
- KIF_BUFFER_NAME — Provides the name of the buffer that will provide or receive data
- KIF_STATUS_BLOCK — Indicates the user control block

10.4.2 KEY$FILE Declarations

The following example shows the required type and procedure formats used to call KEY$FILE:

```
TYPE
    T_KIF_STATUS_BLOCK = PACKED RECORD
    KIF_STATUS_CODE: 0..255;      { RETURN CODE }
    KIF_KEY_NUMBER : 0..255;      { KEY # TO BE ACCESSED }
    KIF_ACCESS     : 0..255;      { FILE ACCESS FLAGS }
    KIF_RESERVED   : 0..255;      { RESERVED—DO NOT MODIFY }
    KIF_RECORD_LEN : INTEGER;  { RECORD SIZE—IF 0, = LRECL }
    KIF_PATHNAME   :  PACKED ARRAY[1..30] OF CHAR;  { FILE NAME }
    END; { T_KIF_STATUS_BLOCK }

PROCEDURE KEY$FILE
    ( KIF_COMMAND              : INTEGER;
    UNIV KIF_KEYNAME           : PACKED ARRAY[1..?] OF CHAR;
    UNIV KIF_BUFFER_NAME       : PACKED ARRAY[1..?] OF CHAR;
    VAR KIF_STATUS_BLOCK       : T_KIF_STATUS_BLOCK); EXTERNAL;
```

10.4.3 KEY$FILE Command Codes

Table 10-1 indicates all valid commands to KEY$FILE. Any code other than those specified yields a status code >F2 (invalid command). In Table 10-1, SVC operation codes and status codes are hexadecimal numbers.

Table 10-1. KEY$FILE Command Codes

Command	Command Code	Operation	SVC Opcode
KIF_INIT	01	Initializes key indexed file handler function. Status codes: none	—
KIF_TERM	02	Terminates key indexed file handler. Status codes: none	—
KIF_OPEN	03	Opens specified file. Status codes: 21, 27, 3B 72, 92, 9A, D0, F6	40
KIF_CLOSE	04	Closes specified file. Status codes: F0	01
KIF_READ_FILE_CHAR	05	Returns file characteristics in KIF_BUFFER_NAME. Status code: 03, F4	05
KIF_SET_ACCESS	06	Sets file access to state specified in status block. Status code: 3B, F5	11
KIF_READ_GREATER	07	The KIF record that has the next value greater than that of the key specified is read into KIF_BUFFER_NAME. Status codes: 05, B4, B8, BF, D7, F0	41
KIF_READ_GREATER_LOCKED	08	Same as KIF_READ_GREATER except record is locked after a successful read. Status codes: 05, B4, B7, B8, D7, F0	41
KIF_READ_BY_KEY	09	The KIF record specified by KIF_KEY_NAME is read into KIF_BUFFER_NAME. Status codes: 05, B4, B5, BF, D7, F0	42
KIF_READ_BY_KEY_LOCKED	10	Same as KIF_READ_BY_KEY except record is locked after a successful read. Status codes: 05, B4, B5, B7, BF, D7, F0	42

Table 10-1. KEY$FILE Command Codes (Continued)

Command	Command Code	Operation	SVC Opcode
KIF_READ_CURRENT	11	Same as KIF_READ_BY_KEY except that pointer set by the set currency commands is used instead of KIF_KEY_NAME. Status codes: 05, B4, B5, D7, F0	42
KIF_READ_CURRENT_LOCKED	12	Same as KIF_READ_CURRENT except record is locked after a successful read. Status codes: 05, B5, B7, D7, F0	42
KIF_READ_GR_OR_EQ	15	Same as KIF_READ_GREATER except that record selected will be equal to or greater than KIF_KEY_NAME. Status codes: 05, B4, B8, BF, D7, F0	44
KIF_READ_GR_OR_EQ_LOCKED	16	Same as KIF_READ_GR_OR_EQ except record is locked after a successful read. Status codes: 05, B4, B7, B8, BF, D7, F0	44
KIF_READ_NEXT	17	Reads the next sequential record as indicated by the currency pointers. Status codes: 05, B3, B7, BD, D7, F0	45
KIF_READ_NEXT_LOCKED	18	Same as KIF_READ_NEXT except record is locked after a successful read. Status codes: 05, B3, B7, BD, D7, F0	45
KIF_INSERT_RECORD	19	Record specified by keyname will be written into file using the contents of KIF_BUFFER_NAME. Status codes: 05, B1, B2, B4, D7, E0, F0	46
KIF_REWRITE_RECORD	20	Replaces the record previously read and locked with the contents of KIF_BUFFER_NAME. Status codes: 05, B2, B5, B9, BA, BE, E0, F0	47

Table 10-1. KEY$FILE Command Codes (Continued)

Command	Command Code	Operation	SVC Opcode
KIF_REWRITE_REC_UNLOCK	21	Same as KIF_REWRITE_RECORD except that the record is unlocked at the completion of the command. Status codes: 05, B2, B5, B9, BA, BE, D7, E0, F0	47
KIF_READ_PREVIOUS	22	Same as KIF_READ_GREATER except that the record accessed has a key value that immediately precedes the one specified by the currency pointers. Status codes: 05, B3, BD, D7, F0	48
KIF_READ_PREVIOUS_LOCKED	23	Same as KIF_READ_PREVIOUS except that the record is locked after a successful read. Status codes: 05, B3, B7, BD, D7, F0	48
KIF_DELETE_BY_KEY	24	Deletes record specified by KIF_KEY_NAME. Status codes: 05, B4, B5, B7, F0	49
KIF_DELETE_CURRENT_REC	25	Deletes record specified by the currency pointers. Status codes: 05, B5, B7, F0	49
KIF_UNLOCK_RECORD	26	Unlocks record specified by the currency pointers. Status codes: B5, B6, B7, BA, F0	4A
KIF_SET_CURRENT_EQUAL	27	Sets currency pointers to the record specified by KIF_KEY_NAME. Status codes: 05, B4, B8, D7, F0	50
KIF_SET_CURRENT_GR_OR_EQ	28	Sets currency pointers at or past the record specified by KIF_KEY_NAME. Status codes: 05, B4, B8, D7, F0	51
KIF_SET_CURRENT_GREATER	29	Sets the currency pointers past the record specified by KIF_KEY_NAME. Status codes: 05, B4, B8, D7, F0	52

Table 10-1. KEY$FILE Command Codes (Continued)

Command	Command Code	Operation	SVC Opcode
KIF_READ_PRIMARY_KEY	30	Same as KIF_READ_BY_KEY except that key 1 is always used as the key number. Status codes: 05, B4, B7, B8, BF, D7, F0	43
KIF_READ_PRIMARY_LOCKED	31	Same as KIF_READ_PRIMARY_KEY except record is locked after a successful read. Status Codes: 05, B4, B5, B7, BF, D7, F0	43

Although not all parameters are used on all calls, all parameters must be provided in each call. Table 10-2 indicates which parameters are actually used in each call. Since all calls use the KIF_RESERVED byte in the KIF_STATUS_BLOCK, Table 10-2 does not include this byte. A Y under a parameter means the command uses that parameter. An N means it does not.

Table 10-2. Parameters Used in KEY$FILE Commands

Command	Key Name	Buffer Name	Key Number	Status Block Pathname	Access
KIF_INIT	N	N	N	N	N
KIF_TERM	N	N	N	N	N
KIF_OPEN	N	N	N	Y	N
KIF_CLOSE	N	N	N	N	N
KIF_READ_FILE_CHAR	N	Y	N	N	N
KIF_SET_ACCESS	N	N	N	N	Y
KIF_READ_GREATER	Y	Y	Y	N	N
KIF_READ_GREATER_LOCKED	Y	Y	Y	N	N
KIF_READ_BY_KEY	Y	Y	Y	N	N
KIF_READ_BY_KEY_LOCKED	Y	Y	Y	N	N
KIF_READ_CURRENT	N	Y	N	N	N
KIF_READ_CURRENT_LOCKED	N	Y	N	N	N
KIF_READ_GR_OR_EQ	Y	Y	Y	N	N
KIF_READ_GR_OR_EQ_LOCKED	Y	Y	Y	N	N
KIF_READ_NEXT	N	Y	N	N	N
KIF_READ_NEXT_LOCKED	N	Y	N	N	N
KIF_INSERT_RECORD	Y	Y	Y	N	N
KIF_REWRITE_RECORD	N	Y	N	N	N
KIF_REWRITE_REC_UNLOCK	N	Y	N	N	N
KIF_READ_PREVIOUS	N	Y	N	N	N
KIF_READ_PREVIOUS_LOCKED	N	Y	N	N	N
KIF_DELETE_BY_KEY	Y	N	Y	N	N
KIF_DELETE_CURRENT_REC	N	N	N	N	N
KIF_UNLOCK_RECORD	N	N	N	N	N
KIF_SET_CURRENT_EQUAL	Y	N	Y	N	N

Table 10-2. Parameters Used in KEY$FILE Commands (Continued)

Command	Key Name	Buffer Name	Key Number	Status Block Pathname	Access
KIF_SET_CURRENT_GR_OR_EQ	Y	N	Y	N	N
KIF_SET_CURRENT_GREATER	Y	N	Y	N	N
KIF_READ_PRIMARY_KEY	Y	Y	N	N	N
KIF_READ_PRIMARY_LOCK	Y	Y	N	N	N

10.4.4 Status Codes Returned to the Program

Table 10-3 lists the hexadecimal codes that are returned in the KIF_STATUS_CODE portion of the KIF_STATUS_BLOCK. The codes are grouped into nonfatal information codes and fatal error codes. The information codes are expected in the normal handling of key indexed files. However, fatal error codes indicate that a severe problem has occurred and the program should probably be aborted. Error codes F0 and greater are generated by KEY$FILE; the others are returned by the operating system. If any error codes occur but are not listed below, consult the *DNOS Messages and Codes Reference Manual*. Look up the status code in the table of internal SVC message codes.

Table 10-3. KEY$FILE Status Codes

Error	Status Code (Hexadecimal)
Information Codes:	
Operation successful	0
No more records	B3
Duplicate found	B4
Record not in file	B5
Record locked	B7
No record satisfies condition	B8
Cannot find next record	BD
Fatal Errors:	
Parameter not in map	05
Invalid device or volume name	21
Nonexistent file	27
Unable to grant access privilege	3B
Not a KIF file	71
Pathname is directory, image, or program file	72
Invalid directory pathname	9A
File inconsistent (reconstruct)	B0
File too large (reconstruct)	B1
Variable-length record too large	B2
Invalid curency information	B6
Attempt to modify nonmodifiable key	B9
Record not locked by requesting task	BA
Out of log blocks	BB
Attempt to rewrite record not in file	BE
Illegal key number	BF
Record too small to contain all of its keys	C0
File is not KIF	D0
Record length must be even	D7
Out of disk space	E0
File not open	F0
KIF command invalid	F2
Buffer size invalid	F4
Access privilege code invalid	F5
Too many files opened	F6

10.4.5 Access Options

The KIF_SET_ACCESS command specifies the type of access desired. It can be executed after a file has been opened. Place one of the following access codes in the KIF_ACCESS portion of the KIF_STATUS_BLOCK:

Code	Meaning
0	Exclusive write
1	Exclusive all
2	Shared
3	Read only

A status code of >3B is returned if the operation cannot be performed.

10.4.6 Restrictions

- The program must specify a buffer whose size is equal to or greater than the logical record length of the file requested. If you request the KIF_READ_FILE_CHAR command, the buffer size must be 68 or more bytes.
- The maximum number of files that you can open without terminating KEY$FILE with a KIF_TERM command is 16.
- There are an additional 40 words of memory allocated for each file opened.
- You must call the initialization routine, KIF_INIT, before using KEY$FILE for the first time. Do not call KIF_INIT more than once during program execution without first terminating the process with a KIF_TERM command.
- If the pathname referenced in the KIF_STATUS_BLOCK is more than 30 characters long, use a synonym.
- Synonyms cannot be used in pathnames when the program is linked with the LUNOBJ library.
- You must maintain a separate status block for each file opened. Do not modify the KIF_RESERVED byte in this block.
- The KIF file used in an open file must already exist. Use the Create Key Indexed File (CFKEY) command to create a key indexed file.
- Files are opened with shared access privileges. To reset the access use the KIF_SET_ACCESS command. KIF_SET_ACCESS returns error code >3B if KIF_SET_ACCESS fails, and error code >F5 if the access code is invalid.

10.4.7 Example Program Using KEY$FILE

The example program in Figure 10-1 shows the initialization, open file, read record, close file, and termination commands of KEY$FILE.

```
PROGRAM KIFXAMPLE
"************************ KIF COMMANDS ***************************
CONST KIF_INIT         = 1;    (* INITIALIE KIF ROUTINE *)
      KIF_TERM         = 2;    (* TERMINATE KIF ROUTINE *)
      KIF_OPEN         = 3;    (* OPEN FILE *)
      KIF_CLOSE        = 4;    (* CLOSE FILE *)
      KIF_READ_BY_KEY  = 9;    (* READ OPERATION *)
TYPE
"************************ KEYED INDEX DATA TYPES ******************
   T_KIF_STATUS_BLOCK       = PACKED RECORD
   KIF_STATUS_CODE          :0..255;   (* RETURN CODE *)
   KIF_KEY_NUMBER           :0..255;   (* KEY # TO BE ACCESSED *)
   KIF_ACCESS               :0..255;   (* FILE ACCESS FLAGS *)
   KIF_RESERVED             :0..255;   (* RESERVED¤DO NOT MODIFY *)
   KIF_RECORD_LEN           : INTEGER; (* REC. SIZE--IF 0, =LRECL *)
   KIF_PATHNAME             : PACKED ARRAY[.1..30.] OF CHAR;
   END; (* T_KIF_STATUS_BLOCK *)
"************************ VARIABLES *******************************
VAR KIF_STATUS_BLOCK      :T_KIF_STATUS_BLOCK;
    KEY                   :PACKED ARRAY[.1..8.] OF CHAR;
    KIF_BUFFER            :PACKED ARRAY[.1..92.] OF CHAR;
"************************ KIF PROCEDURE DEFINITION ******************
PROCEDURE KEY$FILE(KIF_COMMAND:INTEGER;
    VAR KIF_KEYNAME             :PACKED ARRAY[.1..?.] OF CHAR;
    VAR KIF_BUFFER_NAME         :PACKED ARRAY[.1..?.] OF CHAR;
    VAR KIF_STATUS_BLOCK        :T_KIF_STATUS_BLOCK); EXTERNAL;
BEGIN
      (********* INITIALIZE ROUTINE ***********)
   KEY$FILE(KIF_INIT,KEY,KIF_BUFFER,KIF_STATUS_BLOCK);

      (********* OPEN FILE ********************)
   KIF_STATUS_BLOCK.KIF_PATHNAME := 'KEYFILE';
   KEY$FILE(KIF_OPEN,KEY,KIF_BUFFER,KIF_STATUS_BLOCK);
(*>>>====> CHECK KIF_STATUS_BLOCK.KIF_STATUS_CODE *)

     (********* READ RECORD ******************)
   KEY := 'RECORD01';
   KIF_STATUS_BLOCK.KIF_KEY_NUMBER := 1;
   KEY$FILE(KIF_READ_BY_KEY,KEY,KIF_BUFFER,KIF_STATUS_BLOCK);
(*>>>====> CHECK KIF_STATUS_BLOCK.KIF_STATUS_CODE    *)

     (********* CLOSE FILE ******************)
   KEY$FILE(KIF_CLOSE,KEY,KIF_BUFFER,KIF_STATUS_BLOCK);

(********** TERMINATE FUNCTION **********)
   KEY$FILE(KIF_TERM,KEY,KIF_BUFFER,KIF_STATUS_BLOCK);
END.
```

Figure 10-1. Example Program Using Procedure KEY$FILE

10.4.8 Linking KEY$FILE

The file .TIP.MISC.KEY$FI contains the object for the key file routines. Place an INCLUDE command for this file in the link control file for all links that use the key file routines. (Section 7 describes linking procedures.) The source and instructions for KIFTST are on the installation disk, as described in the following paragraph.

10.4.9 KEY$FILE Source

On the TI Pascal object disk, the directory DNPASCAL.TIP.MISC.TIPKIF contains the source for the KIF handling routines. You can change these procedures if they do not meet your requirements. The directory TIPKIF contains one directory and one file. The file, DNPASCAL.TIP.MISC.TIPKIF.BATCH, is a batch stream to compile the key file routines. The directory, DNPASCAL.TIP.MISC.TIPKIF.SRC, contains the source for the key file routines. The files within this directory are as follows:

- PROCES — The process record for the configuration processor that compiles the key file routines.
- KIFTST — The source for a test program that uses the key file routines. This file also explains how to run the test program.

10.5 VDT I/O PROCEDURES

The following paragraphs describe procedures that perform I/O to the VDT. The procedures initialize and clear the screen, display and accept data, and define a filler character.

10.5.1 Procedure Descriptions

All VDT I/O procedures require access to array B, which stores information between calls to the procedures. The procedures are as follows:

INITSCREEN(B,U)
: Initializes the data structure (B) and supplies the unit number (U) for the VDT to which the ACCEPT/DISPLAY calls will be made.This call must precede any other calls. On DNOS, U is a LUNO, and you must assign it before executing the call.

CLEARSCREEN(B)
: Clears the screen.

DISPLAY(B,L,C,S,N)
: Displays string S beginning at line L, column C. The length of the field is the lesser of N or the length of the string S. If the line or column is zero, the field begins at the end of the previous field displayed.

SETFILLER(B,X)
: Use character X as the filler character on subsequent ACCEPT calls. If X is the blank character, the input field will be blank-filled. The null character, CHR(0), disables filling the field and leaves whatever is on the screen in the input field. This is the initial default state.

ACCEPT(B,L,C,S,N,T)
: Read into string variable S from the field beginning at line L, column C. The length of the field is the lesser of N or the length of the variable S. Character variable T is set to the termination character. If the line or column is zero, the field begins at the end of the previous field. The following is a list of character codes returned in T.

Hexadecimal Code	VDT
7F	Erase Field
80	F9
81	F1
82	F2
83	F3
84	F4
85	F5
86	F6
89	Next Field
8A	Next Line
8B	Skip
8C	Home
8D	Return
8E	Erase Input
8F	Initialize Input
93	Enter
94	Previous Field
95	Previous Line
96	F7
97	F8
98	Command
99	Print
9A	F10
9B	Attention
9C	F11
9D	F12
9E	F13
9F	F14

The following is a description of the operations that occur when the specified key is pressed:

Erase Field
: This key fills the field with the fill character and positions the cursor at the beginning of the field.

Erase Input
: This key is equivalent to the Erase Field but returns the character code "#8E" to the calling task.

Home
: This key positions the cursor at the beginning of the field and returns the character code "#8C" to the calling task.

Skip
: This key fills the field from the present cursor position to the end of the field with the fill character, places the cursor one character position beyond the field, and returns the character code "#8B" to the calling task.

Delete Character

This key deletes the character under the cursor and moves all subsequent characters in this field left one character position. Then, the key replaces the last character position of the field with the designated fill character. The position of the cursor remains unchanged. If no characters are present from the cursor position to the end of the field, this key gives a warning beep.

Forward Tab

This key accepts all characters from the present cursor position to the first fill character (if other than a blank) or to the end of the field, leaves the cursor one character position past the end of the field, and returns character code "#89" to the calling task.

Insert Character

If the character under the cursor is the fill character and not a blank, this key gives a warning beep and does nothing. Otherwise, it sets the input mode to insert characters and gives a warning beep when an inserted character causes a character to be lost on the right end of the field. Pressing any key without entering data changes the input mode back to the noninsert mode.

Previous Field

This key positions the cursor to the beginning of the field. If the cursor is at the beginning of the field, this key returns the character code "#94" to the calling task.

Previous Character

If the cursor is at the beginning of the field, this key gives a warning beep. Otherwise, it moves the cursor left one character position.

Previous Line

This key returns the character code "#95" to the task.

Next Character

This key moves the cursor right one character position.

Next Line

This key returns the character code "#8A" to the calling task.

Return

This key acts the same as the Forward Tab key except it returns the character code "#8D".

Enter

This key acts the same as the Forward Tab key except it returns the character code "#93".

10.5.2 Procedure Declarations

Programs that use these routines must include the following set of external procedure declarations:

```
TYPE BTYPE = ARRAY[1..16] OF INTEGER;
PROCEDURE INITSCREEN( VAR BLOCK : BTYPE;
                      UNIT : INTEGER ); EXTERNAL;
PROCEDURE DISPLAY( VAR BLOCK : BTYPE;
                   LINE, COLUMN : INTEGER;
                   BUFFER : PACKED ARRAY[1..?] OF CHAR;
                   LENGTH : INTEGER ); EXTERNAL;
PROCEDURE ACCEPT( VAR BLOCK : BTYPE;
                  LINE, COLUMN : INTEGER;
                  VAR BUFFER : PACKED ARRAY[1..?] OF CHAR;
                  LENGTH : INTEGER;
                  VAR TERMINATIONCHARACTER : CHAR );
                  EXTERNAL;
PROCEDURE CLEARSCREEN( VAR BLOCK : BTYPE );
                       EXTERNAL;
PROCEDURE SETFILLER( VAR BLOCK : BTYPE; EXTERNAL
                     PROCEDURE SETFILLER( VAR BLOCK : BTYPE;
                     C : CHAR ); EXTERNAL;
```

The minimum array size for DNOS is 16.

10.5.3 Linking Procedures

The object file for VDT I/O procedures is .TIP.MISC.ACCDXO. When linking, place an INCLUDE statement for the file .TIP.MISC.ACCDXO in the link control file. (Section 7 describes linking procedures.)

10.5.4 VDT I/O Source

On the TI Pascal object disk, the directory DNPASCAL.TIP.MISC contains the source for the VDT I/O procedures. You can change these procedures if they do not meet your requirements. The source file for DNOS is DNPASCAL.TIP.MISC.ACCDXS. This file contains Pascal source for DNOS VDT I/O routines.

10.6 ADDITIONAL I/O ROUTINES

The following routines allow you to specify an access name or logical unit number (LUNO) to a file, set special operating system file flags, and determine the device type assigned to a file.

10.6.1 Procedure SET$ACNM

Procedure SET$ACNM specifies the access name for a file. Call procedure SET$ACNM to override the default synonym before calling the RESET, REWRITE, or EXTEND procedure that opens the file. SET$ACNM serves the same purpose as procedure SETNAME (see Section 10 of the *TI Pascal Reference Manual*) except that SETNAME specifies a synonym to be mapped into an access name, whereas SET$ACNM specifies the actual access name. To use SET$ACNM, declare type SCI_STRING as in paragraph 10.3.1 and declare the procedure externally.

The declaration for procedure SET$ACNM is as follows:

```
PROCEDURE SET$ACNM (VAR F:<ft>; VAR ACNM: SCI_STRING); EXTERNAL;
```

The <ft> (file type) entry in the PROCEDURE declaration can be either TEXT or a user-defined file type. The first parameter is a file identifier, and the second is a string that contains the access name. The first element of the string is a set to a binary value indicating the length of the string. The access name parameter cannot be a synonym. SET$ACNM associates the specified access name with the specified file.

When a routine that uses LUNO I/O calls procedure SET$ACNM, SET$ACNM associates the access name with the specified file and causes the RESET, REWRITE, or EXTEND operation that opens the file to assign the LUNO to the access name. Unless you have defined a LUNO for the file using the SETLUNO procedure, the default LUNO is used. When defining LUNOs, be sure that two files that are open simultaneously in a program do not use the same LUNO. When the RESET, REWRITE, or EXTEND operation assigns the LUNO, the CLOSE operation releases the LUNO. A LUNO assigned by the task is a task local LUNO under DNOS.

You can use procedure FIND$SYN to identify a synonym and obtain its value for use in a call to SET$ACNM. The following example shows the use of both procedures:

```
CONST SL = 80;                                          (*maximum string length*)
TYPE SCI_STRING = PACKED ARRAY [0..SL] OF CHAR;         (*required type*)
VAR SYN:SCI_STRING;                                     (*for synonym parameter*)
    ACCESS_NAME: SCI_STRING;                            (*for value parameter*)
    DATA_FILE: TEXT;                                    (*for file parameter*)
    N: INTEGER;                                         (*for size of synonym*)
              .
              .
              .
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE: SCI_STRING); EXTERNAL;
                                                        (*external declarations*)
PROCEDURE SET$ACNM (VAR F: TEXT; VAR ACNM: SCI_STRING); EXTERNAL;
            .
            .
            .
BEGIN
            .
            .
            .
RESET(INPUT);                                           (*open input file*)
N:=1;                                                   (*initialize index*)
WHILE NOT EOLN DO BEGIN                                 (*read synonym*)
READ(SYN[N]);
N:=N+1
END;
SYN[0]:=CHR(N-1);                                       (*set length of synonym*)
ACCESS_NAME[0]:=CHR(SL);                                (*set maximum length*)
FIND$SYN(SYN,ACCESS_NAME);                              (*call FIND$SYN*)
IF ORD(ACCESS_NAME[0])=0 THEN                           (*if not synonym*)
        FOR K:=0 TO N DO                                (*copy to access name*)
            ACCESS_NAME[K]:=SYN[K];
SET$ACNM(DATA_FILE, ACCESS_NAME);                       (*set file access name*)
RESET(DATA_FILE);                                       (*open file*)
READ(DATA_FILE,. . . .                                  (*read file*)
```

As seen in the example, byte 0 of string SYN must contain the actual length of the synonym, which may be less than the actual string length. Also, the access name string must have its length placed in byte 0 (remember that access names are limited to 48 characters).

10.6.2 Procedure SETLUNO

The predefined procedure SETLUNO defines a logical unit number for a file in a task that uses LUNO I/O. LUNOs defined by procedure SETLUNO replace the default LUNOs, which are the ASCII code of the first letter of the file name &>41 through >5A and >24). If using SETLUNO, call SETLUNO before calling RESET, REWRITE, or EXTEND to open the file. The syntax for the procedure call is as follows:

```
SETLUNO(<file>,<unit number>)
```

The parameters for the call are file identifier and unit number, an integer value in the range of 0 through 254. SETLUNO only defines the LUNO to be used to access the file; it does not assign the LUNO. Using SET$ACNM (see paragraph 10.6.1), you can have the LUNO assigned automatically, or you can assign a LUNO by using an SCI command prior to executing the task. When a task using SCI synonym I/O calls SETLUNO, no operation is performed.

The following example shows the use of procedure SETLUNO to define LUNO hexadecimal 6E for file FILE2:

```
SETLUNO(FILE2, #6E);
```

10.6.3 Function DEV$TYPE

Function DEV$TYPE returns the device type assigned to a file. Table 10-4 lists the device type codes. To use DEV$TYPE, declare the function externally, as follows:

```
FUNCTION DEV$TYPE(VAR F: TEXT) : INTEGER;EXTERNAL;
```

The following example shows the use of function DEV$TYPE:

```
IF DEV$TYPE(OUTPUT)>0 THEN WRITELN(OUTPUT)
```

Table 10-4. Device Type Codes

Hexadecimal Device Type	Device
0	Dummy device
1	Teleprinter
2	Line printer
3	Cassette
4	Card reader
5	Video display terminal (VDT)
6	Disk
7	Communication device
8	Magnetic tape
FF	Disk file
8000	Symmetric IPC Channel
8001	Master/Slave IPC Channel

10.6.4 Procedure FILE$FLAGS

Use the FILE$FLAGS procedure to read directories as random files or to create temporary files. Declare FILE$FLAGS as follows:

```
PROCEDURE FILE$FLAGS (VAR F: <ft>; USAGE: INTEGER;
                      TEMPORARY: BOOLEAN ); EXTERNAL;
```

where:

<ft> is any file type.

You can call FILE$FLAGS before opening a file to set some of the operating system file flags in the supervisor call (SVC) block. The parameters are as follows:

Parameter	Description
F	The file being referenced
USAGE	Indicates one of the following file usage codes:

Code	Meaning
0	No special usage
1	Directory file
2	Program file
3	Image file

TEMPORARY — Specifies the temporary file flag, as follows:

Value	Meaning
TRUE	Creates a temporary file that is automatically deleted when closed.
FALSE	Creates a normal file that remains after task termination.

The call FILE$ FLAGS(F,0,FALSE) specifies the default conditions that apply when FILE$FLAGS is not used.

10.6.5 Function SCB$A

The function SCB$A returns the address of the SVC call block used by a TIP file. Declare SCB$A as follows:

```
FUNCTION SCB$A( VAR F: <ft> ): SCB__POINTER; EXTERNAL;
```

In the declaration, <ft> represents any file type, and the result is a pointer to a record representing the SVC block. This can be useful when performing operations that use operating system features that are not supported by TIP I/O. The following example illustrates the use of SCB$A:

```
TYPE BYTE = 0..255;
     SCB = PACKED RECORD
                SVC_CODE: BYTE;
                STATUS_CODE: BYTE;
                OP_CODE: BYTE;
                LUNO: BYTE;
                  .
                  .
                  .
                END;
     SCB_POINTER = @SCB;
VAR  P: SCB_POINTER;
     MYFILE: TEXT;
FUNCTION SCB$A(VAR F: TEXT): SCB_POINTER; EXTERNAL;
BEGIN
       .
       .
       .

     P := SCB$A(MYFILE);        (*find SVC block*)
     WITH S=P@ DO
       .
       .                        (*access fields in SVC block*)
       .
END;
```

10.7 OVERLAY LOADER — PROCEDURE OVLY$

The TIP software includes the procedure OVLY$ as the overlay handler. Call procedure OVLY$ in the same module that calls the overlay but prior to calling the procedure within the overlay. Declare OVLY$ in the main program, as follows:

```
PROCEDURE OVLY$(LOAD: INTEGER);EXTERNAL;
```

Initialize the handler with the following call:

```
OVLY$(0);
```

This call must precede any call to OVLY$ to load an overlay. The argument 0 initializes the handler. In subsequent calls to OVLY$, the argument is the overlay number and the call loads the overlay.

Procedure OVLY$ maintains a variable that contains the number of the most recently loaded overlay and compares the argument in the call with that variable. A subsequent call to load the most recently loaded overlay is ignored.

10.8 IDENTIFICATION FUNCTIONS

The identification functions return a requested identifier to the calling task. Function TASKID returns the run-time task identifier, and function STATIONID returns the identifier of the station (terminal) associated with a task.

10.8.1 Function TASKID

Function TASKID obtains the run-time identifier assigned to the task by DNOS. To use TASKID, declare type BYTE and declare function TASKID externally.

The declaration for type BYTE is as follows:

```
TYPE BYTE = 0..#FF;
```

The declaration for function TASKID is as follows:

```
FUNCTION TASKID( ):BYTE; EXTERNAL;
```

There is no parameter for the function. The calling task is assumed to be the task for which the identifier is requested. The following is an example of the use of function TASKID:

```
IDENT: = TASKID;
```

10.8.2 Function STATIONID

Function STATIONID obtains the station identifier of the station associated with the task. To use STATIONID, declare type BYTE and declare function STATIONID externally.

The declaration for function STATIONID is as follows:

```
FUNCTION STATIONID( ):BYTE; EXTERNAL;
```

This function has no parameter; the calling task is assumed to be the task for which the station identifier is requested. The following is an example of the use of function STATIONID:

```
TRMNL: = STATIONID
```

10.9 TIME AND DATE PROCEDURES

The time and date procedures supply time and date information and time delay suspensions to the task. Procedure ITIME returns integer values representing the hour, minute, and second. Procedure IDATE returns integer values representing the year, month, and day of the month. Procedure DELAY places the calling task in a time delay suspension for a specified period. These are in addition to the predefined procedures TIME and DATE, which return the time and date as character strings.

10.9.1 Procedure ITIME

Procedure ITIME obtains the time of day in a record declared for the hour, minute, and second values. To use procedure ITIME, declare record TIME_TYPE and declare procedure ITIME externally.

The declaration for record TIME_TYPE is as follows:

```
TYPE TIME_TYPE = RECORD
                   HOUR:    0..23;
                   MINUTE:  0..59;
                   SECOND:  0..59
                 END;
```

The declaration for procedure ITIME is as follows:

```
PROCEDURE ITIME(VAR TIME: TIME_TYPE); EXTERNAL;
```

The parameter for the procedure is a record into which the procedure places integer values for the hour, minute, and second. The following is an example of a call to procedure ITIME, assuming that variable NOW is a record of type TIME_TYPE:

```
ITIME(NOW);
```

10.9.2 Procedure IDATE

Procedure IDATE obtains the date in a record declared for the year, month, and day values. To use procedure IDATE, declare record DATE_TYPE and declare procedure IDATE externally.

The declaration for record DATE_TYPE is as follows:

```
TYPE DATE_TYPE = RECORD
                   YEAR:    INTEGER;
                   MONTH:   1..12;
                   DAY:     1..31
                 END;
```

The declaration for procedure IDATE is as follows:

```
PROCEDURE IDATE(VAR DATE: DATE_TYPE);EXTERNAL;
```

The parameter for the procedure is a record into which the procedure places integer values for the year, month, and day of the month. The following is an example of a call to procedure IDATE, assuming that variable TODAY is a record of type DATE_TYPE:

```
IDATE(TODAY);
```

10.9.3 Procedure DELAY

Procedure DELAY suspends the calling task for a period of time that is a multiple of 50 milliseconds. The interval is specified as a number of milliseconds, which is rounded off to the nearest 50-millisecond multiple. Declare procedure DELAY externally, as follows:

```
PROCEDURE DELAY(MILLISECONDS: LONGINT);EXTERNAL;
```

The parameter for the procedure is an integer representing a number of milliseconds. The procedure rounds this value to the nearest multiple of 50 milliseconds and suspends the calling task for that interval. The following is an example of a call to procedure DELAY that suspends the task for 100 milliseconds:

```
DELAY(85);
```

10.10 TASK CONTROL PROCEDURES

The task control procedures facilitate cooperation between tasks providing a means of bidding tasks, suspending the calling task, and terminating unconditional suspension of a task. Procedure BID initiates execution of another task. Procedure SUSPEND suspends the calling task unconditionally. Procedure ACTIVATE terminates the unconditional suspension of the specified task, causing the task to resume execution.

10.10.1 Procedure BID

Procedure BID initiates execution of a specified task and passes parameters to the task. To use procedure BID, declare type BYTE and declare procedure BID externally.

The declaration for type BYTE is as follows:

```
TYPE BYTE = 0..#FF;
```

The declaration for procedure BID is as follows:

```
PROCEDURE BID (PROGRAM_FILE_LUNO: BYTE;
               INSTALLED_TASK_ID: BYTE;
               PARAMETERS: LONGINT;
               VAR RUN_TIME_TASK_ID: BYTE;
               VAR STATUS: INTEGER);EXTERNAL;
```

The five parameters for the procedure are as follows:

- LUNO of the program file on which the task is installed. Since task-local LUNO #FF is automatically assigned to a task's program file, LUNO #FF can be used to bid another task on the same program file.
- Installed task identifier
- Parameters for the task (any type having a length of 4 bytes, or two parameters of 2 bytes each)
- A parameter into which procedure BID places the run-time ID of the task
- A parameter into which procedure BID places a status code

When the task being bid is a TIP task linked without library .TIP.MINOBJ, the parameters specify stack and heap requirements. The first two bytes specify stack size in bytes, and the second two bytes specify heap size in bytes. When the task being bid is linked with library .TIP.MINOBJ and begins execution in a procedure (main routine omitted), the parameters are passed to the called procedure and are accessible to the procedure by the type declared in the procedure declaration.

The status code is set to zero to indicate successful bidding of the task. Otherwise, the status code indicates an error code that is defined for the operating system. The DNOS error codes range from >2B01 through >2BFE.

The following is an example of a call to procedure BID to initiate task >1D on the program file assigned to LUNO >3F. The task requires >800 bytes of stack and >200 bytes of heap. TSKID and TSTAT are variables into which the run-time identifier and status are to be stored:

```
BID(#3F,#1D,#08000200,TSKID,TSTAT);
```

The task being bid does not have access to SCI. If it is a TIP program, it must be linked with the LUNOBJ library. If it is an assembly language program, it cannot call any of the S$... routines.

10.10.2 Procedure SUSPEND

Procedure SUSPEND places the calling task in unconditional suspension. To use procedure SUSPEND, declare the procedure externally, as follows:

```
PROCEDURE SUSPEND; EXTERNAL;
```

The procedure has no parameter. It assumes that the calling task is the task to be suspended. The following is an example of a call to procedure SUSPEND:

```
SUSPEND;
```

10.10.3 Procedure ACTIVATE

Procedure ACTIVATE terminates unconditional suspension of a specified suspended task. To use procedure ACTIVATE, declare type BYTE and declare procedure ACTIVATE externally.

The declaration for procedure ACTIVATE is as follows:

```
PROCEDURE ACTIVATE(RUN_TIME_TASK_ID:BYTE;
                   VAR STATUS:INTEGER ); EXTERNAL;
```

The parameters are the run-time task identifier and a parameter into which the procedure places a status code. The run-time task identifier is not necessarily the installed task identifier; it is returned by a call to procedure BID when the task is initiated by procedure BID. Each task can obtain its own run-time task identifier by calling function TASKID. When the activation of the task is successful, the procedure returns zero in the variable named as the second (status) parameter. Otherwise, the procedure returns a system error code (>0700 through >07FF).

The following is an example of a call to activate the task whose run-time identifier is >3C, placing the status in variable TSTAT:

```
ACTIVATE(#3C,TSTAT);
```

10.11 MESSAGE-HANDLING PROCEDURES

The message-handling procedures send and receive messages between tasks and delete queued messages. Procedure PUTMSG places a message in a message queue. Procedure GETMSG obtains the next message from a queue. Procedure PRGMSG purges a message queue of any undelivered messages.

These procedures are primarily intended for use under DX10. The message-handling SVCs used by these procedures are optional in a DNOS system generation. Use IPC channels instead of these procedures in application programs used only under DNOS. You can write to and read from a symmetric IPC channel as you would a Pascal sequential or text file.

10.11.1 Procedure PUTMSG

Procedure PUTMSG places a specified message on a specified message queue. To use procedure PUTMSG, declare type BYTE and declare procedure PUTMSG externally.

The declaration for type BYTE is as follows:

```
TYPE BYTE = 0..#FF;
```

The declaration for procedure PUTMSG is as follows:

```
PROCEDURE PUTMSG  (QUEUE:BYTE;
                   UNIV MESSAGE:PACKED ARRAY[1..?] OF CHAR;
                   VAR STATUS:BYTE);EXTERNAL;
```

The queue parameter is an arbitrarily assigned number that identifies a message queue. A convenient method of assigning queue numbers that ensures a unique number is to use the run-time identifier of the task to which the message is directed as the queue number. When a task initiates another task by calling the BID procedure, the run-time identifier of the initiated task is returned. A task can obtain its own run-time identifier by calling function TASKID.

The message parameter is the variable that contains the literal message to be placed in the queue. The status parameter is a variable into which the procedure places the status code. The status code is zero when the message is enqueued successfully. The status is a nonzero value when the queue is full.

The following is an example of a call to procedure PUTMSG; the queue is >3C, the message is contained in variable MSG1, and the status is returned in variable MSTAT:

```
PUTMSG(#3C,MSG1,MSTAT);
```

10.11.2 Procedure GETMSG

Procedure GETMSG obtains the next message from the specified queue. To use procedure GETMSG, declare type BYTE and declare procedure GETMSG externally.

The declaration for procedure GETMSG is as follows:

```
PROCEDURE GETMSG  (QUEUE:BYTE;
                   UNIV BUFFER:PACKED ARRAY[1..?] OF CHAR;
                   VAR LENGTH,STATUS:BYTE);EXTERNAL;
```

The queue parameter is the queue number that other tasks use for messages directed to the calling task. The run-time task identifier is often used; you can obtain it by calling function TASKID. The buffer parameter is the variable into which the procedure places the received message. The length parameter is the variable into which the procedure places the number of bytes in the message. The status parameter is the variable into which the procedure places the status code: zero when the message is received, and nonzero when the queue is empty.

The following is an example of a call to procedure GETMSG; the queue is >3C, the message buffer is variable MSG2, the variable for the length is MLGTH, and the status variable is MSTAT:

```
GETMSG(#3C,MSG2,MLGTH,MSTAT);
```

10.11.3 Procedure PRGMSG

Procedure PRGMSG empties a specified message queue by discarding any messages in the queue. To use procedure PRGMSG, declare type BYTE and declare procedure PRGMSG externally.

The declaration for procedure PRGMSG is as follows:

```
PROCEDURE PRGMSG(QUEUE:BYTE);EXTERNAL;
```

The queue parameter is the queue number that identifies the queue to be purged. The following is an example of a call to procedure PRGMSG for message queue >3C:

```
PRGMSG(#3C);
```

10.12 SYSTEM COMMON ACCESS PROCEDURES

The system common procedures allow access to the system common area of memory defined during generation of the operating system. Under DNOS, system common is not available to tasks that use more than two memory segments for other purposes (for example, a task segment and two procedure segments). Also, when system common is accessed, the heap region cannot expand. Procedure SYSCOM obtains the address and size of the system common area. Procedure RLSCOM releases the system common area from task access.

In a DNOS system, system common SVCs are optional. The system common SVC group must be explicitly included during system generation.

10.12.1 Procedure SYSCOM

Procedure SYSCOM allows access to the system common area by obtaining the address and size of the area defined for the operating system. To use procedure SYSCOM, declare the procedure externally, as follows:

```
PROCEDURE SYSCOM(VAR ADDRESS,LENGTH:INTEGER);EXTERNAL;
```

The two parameters are integer variables into which the procedure places the address and the length, respectively, of the system common area. The following is an example of a call to procedure SYSCOM, specifying variables ADDR and LEN for the address and length:

```
SYSCOM(ADDR,LEN);
```

10.12.2 Procedure RLSCOM

Procedure RLSCOM releases the task from access to system common. To use procedure RLSCOM, declare the procedure externally, as follows:

```
PROCEDURE RLSCOM( );EXTERNAL;
```

The procedure has no parameter. The following is an example of a call to procedure RLSCOM:

```
RLSCOM;
```

10.13 PROCEDURE INIT$BLOCK

You can efficiently initialize an entire array or record by using the procedure INIT$BLOCK. Declare INIT$BLOCK externally, as follows:

```
PROCEDURE INIT$BLOCK (LOC, DIX, VAL:INTEGER); EXTERNAL;
```

The parameters are the starting location, the size of the area to be initialized (number of bytes), and the value to be placed in each word of the area. The following initializes array X to zeros:

```
INIT$BLOCK(LOCATION(X), SIZE(X), 0);
```

Using INIT$BLOCK together with the INITCOMMON compiler option, you can specify initial values for COMMON variables. (Compiler options are described in Section 11 of the *TI Pascal Reference Manual*.)

10.14 PROCEDURE SVC$

Use procedure SVC$ to execute any SVC of the operating system under which the task is executing. To use SVC$, declare the structure for the SVC block and declare procedure SVC$ externally. Next, call procedure NEW to obtain an area for the SVC block and then build the SVC block. Then, call procedure SVC$.

The following is an example of the declarations required for an SVC block and for a pointer to the SVC block:

```
TYPE POINT = @DANDT;
    SCB = RECORD
            CODE:INTEGER;
            TIME:POINT
          END;
    PT = @SCB;
    DANDT = ARRAY[1..5] OF INTEGER;
VAR BLOCK : PT;
```

The declaration for procedure SVC$ is as follows:

```
PROCEDURE SVC$(P:PT);EXTERNAL;
```

The SVC block in the example is the block for a date and time supervisor call. This SVC requires a four-byte block with the code in the first byte and zero in the second byte; the address of a five-word array for the date and time is in the third and fourth bytes. By declaring a record consisting of an integer and a pointer, you can initialize the code and the zero by an assignment. You can obtain the array by a call to procedure NEW, placing the array's address in the record. You can also obtain the record SCB by a call to procedure NEW, placing the array's address in the record. You can also obtain the record SCB by a call to procedure NEW, assigning its address to pointer PT. The following example shows the calls to procedure NEW and the call to procedure SVC$:

```
NEW(BLOCK)                  (*obtain SVC block*)
WITH BLOCK@ DO BEGIN
NEW(TIME)                   (*obtain array for date and time*)
CODE: = #0300;              (*assign code and zero*)
SVC$(BLOCK);                (*get date and time*)
END;
```

The result of the example call is that the system has placed the year in element 1 of the array to which TIME points, the day in element 2, the hour in element 3, the minute in element 4, and the second in element 5. All values are binary; the year is the binary equivalent of the two least-significant digits of the year.

Alternatively, you can declare procedure SVC$ to use a VAR parameter that is the SVC block. By this method, the preceding example can be executed as follows:

```
TYPE POINT  = @DANDT;
     SCB    = RECORD
                CODE: INTEGER;
                TIME: POINT;
                END;
     DANDT = ARRAY[1..5]OF INTEGER;
VAR  SBLOCK: SCB;
PROCEDURE SVC$(VAR B:SCB); EXTERNAL;
        .
        .
        .
     WITH SBLOCK DO BEGIN
          NEW(TIME);
          CODE : = #0300;
          SVC$(SBLOCK);
          END;
```

10.15 SEMAPHORE PROCEDURES

The semaphore procedures support the use of semaphores to synchronize execution of cooperating tasks. A semaphore is an integer variable, typically a field of a COMMON block, shared by several tasks (see paragraph 7.3.9.1). The typical use of a semaphore is to coordinate execution of critical code. Critical code is code that cannot be executed until another cooperating task executes, or code that must be executed before a cooperating task can execute part (or all) of its code.

The semaphore has two states, set and reset. Initially, the semaphore is reset by the task that performs initialization. When a section of code in a cooperating task is critical, the task tests the semaphore to determine if the critical code can be executed. When an activity of another task has set the semaphore to prevent any task from executing critical code, the task continues to test the semaphore before executing the critical code. When the semaphore is reset, the task sets the semaphore and executes the critical code. When execution of the critical code is completed, the task resets the semaphore. If another task has been awaiting the resetting of the semaphore, that task can now continue. Procedure RESETSEMAPHORE resets a specified semaphore. Procedure TESTANDSET tests a semaphore, returns the state of the semaphore, and sets the semaphore if it was reset when tested.

10.15.1 Procedure RESETSEMAPHORE

To use procedure RESETSEMAPHORE, declare the procedure externally as follows:

```
PROCEDURE RESETSEMAPHORE(VAR SEMAPHORE:INTEGER);EXTERNAL;
```

The parameter is the semaphore variable to be reset. An example of the use of the procedure is included in Figure 10-2

10.15.2 Procedure TESTANDSET

Procedure TESTANDSET tests a specified semaphore and sets a Boolean variable to the opposite state. When the semaphore is reset, the procedure also sets the semaphore. To use procedure TESTANDSET, declare the procedure externally, as follows:

```
PROCEDURE TESTANDSET  (VAR SEMAPHORE:INTEGER;
                       VAR SET_OK:BOOLEAN);EXTERNAL
```

The first parameter is an integer variable containing a value the procedure tests. The second parameter is a Boolean variable that is set to FALSE when the tested variable contains a value that is interpreted as the set state. The Boolean variable is set to TRUE when the tested variable contains a value that is interpreted as the reset state. The tested variable (semaphore) is set at the same time that the Boolean variable is set to TRUE.

```
COMMON FLAG:INTEGER;          (*semaphore variable*)
  .
  .
  .
VAR OK:BOOLEAN;               (*TRUE/FALSE variable*)
  .
  .
  .
RESETSEMAPHORE(FLAG);         (*initialize semaphore*)
  .
  .
  .
REPEAT                        (*await permission*)
  TESTANDSET(FLAG,OK);        (*to execute*)
  IF NOT OK THEN DELAY (100); (*critical code*)
UNTIL OK;
  .
  .                           (*critical code here*)
RESETSEMAPHORE(FLAG);         (*reset semaphore after
                                executing critical code*)
```

Figure 10-2. Semaphore Example

Figure 10-2 shows the use of a semaphore to control execution of critical code in a task. The semaphore is integer variable FLAG in the COMMON block. A similar declaration in each of the cooperating tasks makes this variable available to each task. Boolean variable OK is local to the task and contains the result of the test of the semaphore by procedure TESTANDSET.

The task initializes the semaphore by executing the RESETSEMAPHORE procedure. At this point, any cooperating task that executes procedure TESTANDSET to test semaphore FLAG sets the semaphore and sets the Boolean variable parameter to true.

The REPEAT statement encloses both a call to procedure TESTANDSET and an IF statement. Procedure TESTANDSET sets Boolean variable OK to FALSE as long as semaphore FLAG remains set. The IF statement calls procedure DELAY, suspending the task for 100 milliseconds when Boolean variable OK is FALSE. The calls to procedures TESTANDSET and DELAY are repeated until semaphore FLAG is reset. When the task that has set the semaphore completes its critical code and rests the semaphore, procedure TESTANDSET sets the semaphore and sets Boolean variable OK to TRUE. Procedure DELAY is not called, and the code following the REPEAT statement is executed.

The critical code follows the REPEAT statement and ends with a call to procedure RESETSEMAPHORE. This code can be executed only while no other task that uses semaphore FLAG is executing its critical code.

11

Assembly Language Routines

11.1 GENERAL

You can write a procedure or function in assembly language to be used with a TIP program. The procedure or function must comply with the following requirements:

- Format the routine in a manner compatible with the TIP compiler and run-time system.
- Specify the static nesting level of the routine.
- Call the appropriate entry handler for the routine.
- Declare the routine as external in the appropriate declaration section of the calling program.

The format of the routine involves both the format of the data in the stack frame and the format of the routine itself. Also, the format varies depending on the category of the routine. The run-time system groups routines into three categories according to the degree of complexity of the routine. The run-time system includes a separate handler for each category. The more complex routines require a more complex handler; less complex routines can use a simpler handler that requires less time to execute.

This section describes the static nesting level, its relation to the routine, and the categories of routines. It then describes the workspace, system storage, and data areas of the stack frame, and the routine itself.

In some applications, you can provide routines for program error termination. To illustrate the interface with these routines, this section includes descriptions of termination routines in the TIP libraries.

11.2 STATIC NESTING LEVEL

A TIP program consists of a main program that can declare one or more routines. Each routine can also declare one or more routines. The main program is at static nesting level one. Any or all routines declared in the main program are at static nesting level two. Any or all routines declared in a level two routine are at static nesting level three. Generally, the static nesting level of routine X is one greater than that of the routine in which routine X is declared. The run-time support system supports 16 static nesting levels. The static nesting level must be explicitly identified within an assembly language routine for the run-time system to properly interface with the routine.

11.3 ROUTINE CATEGORIES

A routine that neither declares nor calls another routine using Pascal linkage and that requires no more than 64 bytes of stack space is in the short category. A routine that does not declare another routine and that either calls one or more routines or requires more than 64 bytes of stack space is in the medium category. A routine that declares one or more routines that can globally access its variables is in the standard category.

A routine written in assembly language does not actually declare a routine. There is no PROCEDURE or FUNCTION declaration in assembly language. The equivalent of declaring a routine in assembly language is calling a routine that has a static level number greater than that of the calling routine. The category of the routine determines the uses of registers and the entry handler that the routine calls.

11.4 THE STACK FRAME AND THE ROUTINE

The stack frame was described in Section 8. The following paragraphs explain, in greater detail, the workspace, system storage, and data areas of the stack frame as they relate to the routine. For convenient reference, Figure 11-1 reiterates the structure of the stack frame. The stack frame is organized as shown in Figure 11-1.

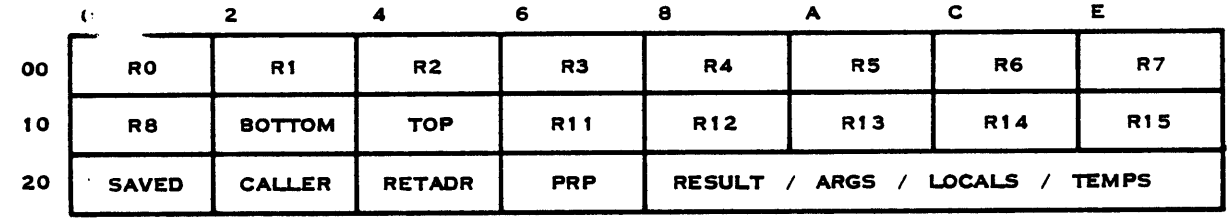


Figure 11-1. Stack Frame Structure

The contents of the stack frame are as follows:

- R0–R15 — Workspace registers.
- BOTTOM — bottom (beginning) of stack frame (dedicated register).
- TOP — Top-of-stack (dedicated register).
- R12 — Display pointer (contains address of display on entry to a procedure).
- R13 — Global values (standard linkage); saved WP (short or medium linkage).
- R14 — Global values (standard linkage); saved PC (short or medium linkage).

- R15 — Global values (standard or medium linkage); saved ST (short linkage).
- SAVED — Saved display pointer (standard linkage). The routine being entered saves the old display pointer at its level in this field prior to storing its bottom-of-stack pointer in the display. Used only by routines containing nested routines.
- CALLER — Caller's WP (standard).
- RETADR — Caller's PC (standard); #0000 (short); #FFFF (medium).
- PRP — Process record pointer; used with LUNOBJ library only.

Variable Length Area

- RESULT — Function result,if any (eight or ten bytes, left-justified).
- ARGS — Arguments.
- LOCALS — Local variables.
- TEMPS — Compiler-generated temporaries.

11.4.1 Workspace

The workspace is the set of workspace registers (R0 through R15) that the routine uses. Entry to the routine is by a BLWP instruction. This places the lowest address in the stack frame in the workspace pointer (WP) register, creating the first 32 bytes of the stack frame the workspace.

For standard category routines, only registers R9 and R10 are dedicated (that is, the contents of these registers must not be altered). Register R9 contains the address of the bottom of the stack frame (the lowest address of the stack frame of the routine). Register R10 contains the address of the top of the stack (the word beyond the word at the highest address in the stack frame of the routine). Other registers can be used as desired. Register R12 contains the address of the process record.

For medium category routines, registers R13 and R14 are dedicated, in addition to the registers used for standard routines. Register 13 contains the calling routine's WP and register R14 contains the return address in the calling routine. Only the computer hardware uses registers R13 and R14. The entry handler does not store the values for restoration by the return handler.

For short category routines, in addition to the registers used for medium and standard category routines, register R15 is dedicated. Register R15 contains the contents of the status register (ST) of the computer at the time of the call to the routine.

11.4.2 System Storage

The entry handler uses the four-word area following the workspace in the stack frame for storage of return information. The use of the words differs for each category of routine.

The run-time system maintains an array labeled DISPLAY in the process record. The array contains the address of the stack frame of static nesting level one in the first element (address 2 relative to the process record address in R12). Successive elements of the array contain the addresses of the stack frames of the current routines at successive nesting levels. That is, element two of the array contains the stack frame address of the most recently called routine at static nesting level two. Element three contains such an address for level three, and so on. The addresses in the DISPLAY array are called *display entries.* The display entries enable a routine to access the variables in all currently active stack frames.

Array DISPLAY contains pointers to stack frames of the most recently called routines at the static nesting levels. However, routines can call other routines at the same level and can also call themselves (recursion). When a routine calls itself or another routine at the same level, dynamic nesting results. The dynamic nesting relationships are implied in the order of stack frames in the stack and in the information stored in the stack frames.

The entry handler for a standard category routine stores the display entry in the first word of the area of the stack frame that follows the workspace, address >20 relative to the address of the stack frame. The entry handler stores the WP register of the calling routine in the following word, address >22, and the return address in the next word, address >24. The last word in this area, address >26 relative to the stack frame address, is not used for tasks that use SCI I/O. Tasks that use LUNO I/O place the current process record address in address >26.

The entry handler for a medium category routine uses only the word at address >24. The handler stores the value −1 in that word, identifying the stack frame as that of a medium category routine. The return information remains in registers R13 and R14, which are not available to the routine.

The entry handler for a short category uses only the word at address >24. The handler stores 0 in that word, identifying the stack frame as that of a short category routine. The return information remains in registers R13, R14, and R15, which are not available to the routine.

11.4.3 Data

The remainder of the stack frame is of variable length and contains the data required by the routine. The first area, the area into which the routine must place the result, applies only to functions. When the type of the function result is DECIMAL, the area is ten bytes in length. Results of other types must be left justified in an eight-byte field.

The second area for functions (the first area for procedures) is the area for parameters or arguments. Parameters are passed by value or by reference in TIP. When a parameter is passed by value, a copy of the parameter is placed in the stack frame immediately prior to the call of the routine. The parameter occupies a variable amount of space determined by the type of the parameter. When a parameter is passed by reference, the address of the parameter is placed in the stack. A parameter passed by reference requires one word in the stack regardless of the type of the parameter.

Two types of parameters require special additional information: parameters with dynamic bounds, and procedure or function parameters. When a parameter with one or more dynamic bounds is passed by reference, the address of the parameter (array or set), the upper bound (or bounds, if an array has more than one dimension with a dynamic upper bound), and the size are placed in the stack frame. When a parameter with dynamic bounds is passed by value, the same information is placed in the stack frame, and a copy of the parameter is placed in the dynamic portion of the stack frame. This requires that the called routine extend the stack frame, copy the parameter into the extended stack frame, and change the address in the stack frame to point to the copy.

FIGURE 11-2 is an example of an assembly language procedure that has a parameter with a dynamic bound. This procedure, P$PARM, is a level 2 routine that returns a specified SCI parameter in array STRING. The length of array STRING is determined by the length of the corresponding parameter in the procedure call. The procedure calls SCI routine S$PARM (described in the *DX10 Operating System Systems Programming Guide, Volume V)* to obtain the parameter. The declaration for the procedure is shown in the first six comment lines except that the keyword EXTERNAL must follow the parameter list. Note that this routine cannot be used with programs linked with the .TIP.LUNOBJ or .TIP.MINOBJ libraries.

```
       IDT 'P$PARM'
* PROCEDURE P$PARM
* (     I: INTEGER; { Sequence number of desired parameter }
*     VAR STRING: { Buffer to hold returned parameter }
*       PACKED ARRAY [0..?] OF CHAR;
*     VAR ERROR: INTEGER { Error indicator: nonzero implies error }
* )
* This routine provides a TIP interface to the SCI990 routine S$PARM.
*   The Ith parameter is fetched from the TCA file if*      (1) it exists, and
*     (2)STRING is large enough to contain it.
*   The length of the parameter is returned in STRING[0];
*   STRING[0]=0 implies that the Ith parameter does not exist.
*   ERROR <> 0 implies that an SCI-detected error has occurred;
*   ERROR is the error code returned by SCI.
*   (ERROR=>901B implies STRING too small to hold result.)
*   Note: STRING need not be preinitialized in any way.
*
*  Register Equates:
BOT     EQU   R9              Start of stack frame
ARG     EQU   >28             Beginning of argument list
*
I       EQU   ARG+0           Value of I
UB      EQU   ARG+2           Upper bound of STRING
SIZE    EQU   ARG+4           Total length of STRING
STRING  EQU   ARG+6           Address of STRING
ERROR   EQU   ARG+8           Address of ERROR
*
        REF   ENT$S,RET$S
        REF   S$PARM
*
L1      TEXT  'P$PARM'
        DATA  2               Level Number
        DATA  L2              Epilogue Number
        DATA  L1              Literals Area
        DEF   P$PARM
P$PARM  BL    @ENT$S
        DATA  >32             Stack Frame Size
*                                     Set up arguments:
        MOV   @I(BOT),R1                Parameter number
        MOV   @STRING(BOT),R2           STRING Address
        MOVB  @UB+1(BOT),*R2            STRING Length (first byte of string)
        BLWP  @S$PARM                 Call S$PARM
        BYTE  R1,R2                     (specify argument locations)
        MOV   @ERROR(BOT),R3          Get error code
        MOV   R0,*R3
L2      B     @RET$S                  Return to caller
        END
```

Figure 11-2. Assembly Language Routine Example

The routine uses EQU directives to define displacements for the parameters. Note that the first parameter, I, is in the first word of the parameter list of the stack frame. The next word contains the upper bound of array STRING, followed by the size and address of the array. The last word of the parameter list of the stack frame contains the address of the third parameter, ERROR.

When a parameter is a procedure or function, the parameter is represented in the stack frame by a structure that contains the entry point address of the procedure and a copy of the set of display pointer values that provides the correct environment for the procedure or function. The environment of the procedure or function passed as a parameter may differ from that of the called routine. The TIP run-time system uses the environment of the parameter routine rather than that of the called routine. This requires that the DISPLAY array containing the environment of the parameter routine be copied into the stack frame. When the parameter routine is executed, the software saves the contents of the DISPLAY array and places the copy from the stack frame into the array. Upon return from the parameter routine, the saved contents of the DISPLAY array are restored.

The parameters in the stack frame are followed by the local variables for the routine. This area is of variable length, determined by the number of variables and the type of each.

When compiling routines written in TIP, the compiler places temporary variables in the stack frame following the local variables. Temporary variables are variables in which the routine stores intermediate results, or similar data. When writing a routine in assembly language, allow space in the stack frame for temporary variables by increasing the stack frame length appropriately and accessing the additional words as required.

11.5 THE ROUTINE MODULE

The module for the routine contains required constant information, constants and literals required by the routine, and the executable code for the routine. Figure 11-3 shows the structure of the module for a standard category routine.

```
literals  TEXT 'ROUTINE'              8 CHARACTER MODULE NAME
          DATA n                      STATIC NESTING LEVEL
          .
          .    constants and literals
          .
          DATA epilogue               (ENTRY POINT - 4)
          DATA literals               (ENTRY POINT - 2)
ROUTINE   EQU  $                      ROUTINE ENTRY POINT
          BL   @ENT$n                 BRANCH TO ENTRY HANDLER
          DATA frame size             SIZE OF STACK FRAME REQUIRED
          .
          .    routine body
          .
epilogue  EQU  $
          .
          .    epilogue code
          .
          B    @RET$n                 BRANCH TO RETURN HANDLER
```

Figure 11-3. Structure of a Standard Category Routine at Level n

The first section of the code for the module contains constants. The first statement must be a TEXT directive to place the module name in the first four words of the module. Fill the operand to the right with blanks when the module name contains fewer than eight characters. The next statement must be a DATA directive to place the static nesting level in the next word.

These constants required by the run-time support system are followed by constants and literals required for the routine. The directives required to define these items, if any, are followed by two more DATA directives. The operand of the first DATA directive is the label of the routine's epilogue. The operand of the second is either the label of the TEXT directive that contains the module name, or zero if the name is omitted.

The constants required by the run-time support system are used in the event of an escape from the module. When a routine does not require any constants and an escape from the module cannot occur, all of the constants previously described may be omitted except for a single DATA directive with an operand of zero, indicating that the constants section is empty.

The entry point of the routine follows the constants section and must be a BL instruction to branch to the appropriate entry handler. The branch of the entry handler is BL @ENT$S for a short category routine, BL @ENT$M for a medium category routine, a BL @ENT$n for a standard category routine, where n is the static nesting level of the routine. A DATA directive with the frame size as the operand follows the BL instruction.

The executable code of the routine follows the DATA directive. Following this code is the epilogue, which performs any required termination functions and calls the return handler. Each category has a return handler corresponding to the entry handler. For a short category routine, the branch to the return handler is B @RET$S. For a medium category routine, it is B @RET$M. And for a standard category routine, it is B @RET$n, where n is the static nesting level of the routine.

11.5.1 ACCESS TO VARIABLES

A TIP routine has access to its local variables, to variables declared in the routine in which that routine was declared, and to variables of routines active at each level, back to the global variables declared in the main program. Access to parameters passed by value and to local variables at each level involves accessing the variable from the stack frame. Access to parameters passed by reference involves accessing the address in the stack frame and accessing the variable at that address.

To access a parameter passed by value or a local variable, use indexed addressing. R9 contains the stack frame address. The following is an example of code to access a value parameter or a local variable:

```
NUM     EQU     >28                 DISPLACEMENT OF PARAMETER
        .
        .
        .
        MOV     @NUM(R9),R0         MOVE PARAMETER TO R0
```

The DISPLAY array in the process record contains the stack frame addresses of all active stack frames. R12 contains the address of the process record upon return from the entry handler. If R12 has been used for other storage, the address of the process record can be restored in R12 as follows:

```
REF     CUR$$                       REFERENCE ROUTINE
.
.
.
BL      @CUR$$                      CALL ROUTINE
```

To obtain the address of the stack frame for a particular level, compute the displacement by multiplying the level number by two. Then use the displacement within the stack frame for a particular variable to access the variable. The following is an example:

```
INT     EQU     >30                 DISPLACEMENT OF VARIABLE INT
*                                   IN LEVEL 2 STACK FRAME
DLEV2   EQU     2*2                 DISPLACEMENT FOR LEVEL 2
*                                   DISPLAY POINTER
        .
        .
        .
        MOV     @DLEV2(R12),R2      MOVE LEVEL 2 STACK FRAME
*                                   ADDRESS INTO R2
        MOV     @INT(R2),R3         MOVE INT TO R3
```

Accessing parameters passed by reference requires accessing the address, then accessing the parameter at the address. The following is an example:

```
NUM     EQU     >28                 DISPLACEMENT OF PARAMETER
        .
        .
        .
        MOV     @NUM(R9),R2         MOVE PARAMETER ADDRESS INTO
*                                   R2
        MOV     *R2,RO              MOVE PARAMETER TO R0
```

Accessing a parameter passed by reference to a routine at another level requires accessing the address from the stack frame at that level, then accessing the parameter at the address. The following is an example:

```
INT     EQU     >30                 DISPLACEMENT OF PARAMETER
*                                   INT IN LEVEL 2 STACK FRAME
DLEV2   EQU     2*2                 DISPLACEMENT FOR LEVEL 2
*                                   DISPLAY POINTER
        .
        .
        .
        MOV     @DLEV2(R12),R2      MOVE LEVEL 2 STACK FRAME
*                                   ADDRESS INTO R2
        MOV     @INT(R2),R4         MOVE PARAMETER ADDRESS INTO R4
*       MOV     *R4,R3              MOVE INT TO R3
```

11.5.2 Calling a Routine

Calling a routine requires making parameters, if any, available to the called routine and transferring control to the entry point of the called routine. Copies of value parameters must be moved into the stack frame of the called routine. Addresses of reference parameters must be placed in the stack frame of the called routine. Register R10 contains the address of the first word beyond the stack frame of the routine; this address becomes that of the stack frame of the called routine. Copies of variables or addresses of parameters can be moved into the stack frame of a called routine by using the displacement for the variable indexed by R10. The following is an example of moving a variable of a routine into the stack frame of the called routine:

```
VAL     EQU     >28                 DISPLACEMENT OF LOCAL
*                                   VARIABLE VAL
NUM     EQU     >2A                 DISPLACEMENT OF VALUE
*                                   PARAMETER FOR CALLED ROUTINE
        .
        .
        .
        MOV     @VAL(R9),@NUM(R10)  MOVE COPY TO NEW STACK FRAME
```

The following is an example of moving a parameter address from a stack frame at a higher level to the stack frame of the called routine:

```
INT     EQU     >30                 DISPLACEMENT OF PARAMETER
*                                   INT IN LEVEL 2 STACK FRAME
DLEV2   EQU     2*2                 DISPLACEMENT FOR LEVEL 2
*                                   DISPLAY POINTER
VAL     EQU     >28                 DISPLACEMENT FOR PARAMETER
*                                   IN CALLED ROUTINE STACK FRAME
        .
        .
        MOV     @DLEV2(R12),R2      MOVE LEVEL 2 STACK FRAME
*                                   ADDRESS INTO R2
        MOV     @INT(R2), @VAL(R10) MOVE PARAMETER ADDRESS TO
*                                   CALLED ROUTINE STACK FRAME
```

Use a BLWP instruction to transfer control to a routine. The transfer vector must be created at run time because the stack frame address is determined at that time. Register R10 contains the stack frame address for the called routine, which is also the workspace address. The code to call a routine is as follows:

```
        REF     ROUTINE
TOP     EQU     R10                 POINTER TO STACK FRAME OF
*                                   CALLED ROUTINE
        .
        .
        .
        LI      TOP+1, ROUTINE      PLACE ENTRY POINT OF ROUTINE
*                                   IN R11
        BLWP    TOP                 BRANCH TO ROUTINE
```

11.6 ALTERNATIVE METHODS

Alternative methods of creating assembly modules are discussed below.

11.6.1 Reverse Assembler

To write a routine in assembly language, write a dummy routine in TIP and compile the program. The dummy routine should declare the routine precisely so that the stack frame is built by the compiler to provide for the parameters and variables required. The statement portion of the routine may approximate the processing required for the routine or may consist only of the BEGIN and END keywords required. The object module that the compiler produces can be submitted to the Reverse Assembler (RASS) to list the assembly language corresponding to the object code. The interface with the run-time system is thus produced by the compiler, and you need only add the assembly language code to perform the processing.

11.6.2 Assembly Language Extractor

The assembly language extractor utility can also be used to produce TIP-compatible assembly language modules. This utility takes as input the assembly language instructions placed in a compiler listing when a routine has been compiled with the LISTOBJ option. Refer to paragraph 5.5.4 for a description of LISTOBJ output, and paragraph 5.3.7 for an expanation of the assembly language extractor utility.

11.7 USER TERMINATION ROUTINES

For special requirements where normal termination routines are inadequate, you can write routines to perform error termination. The following paragraphs describe the TIP library routines and show examples of the code. The termination sequence provided by the standard library (.TIP.OBJ) is described in paragraph 11.7.1. Paragraphs 11.7.2 and 11.7.3 explain the differences that exist when other libraries are used.

11.7.1 Standard Termination Routines

When a TIP task terminates, control passes to library routine TERM$. TERM$ is a Pascal callable routine with the address of the process record for the terminating task as its parameter. TERM$ is part of the initiate/terminate process, a separate process from the process for the user task. The parameter, the address of the task's process record, allows TERM$ to access the contents of the WP, PC, and ST registers in bytes >22 through >27 of the process record. TERM$ can also access the termination code, bytes >3C and >3D of the process record.

TERM$ writes a message that specifies the reason for termination and calls the abnormal termination dump routines when appropriate. It displays the amount of stack and heap used and then calls P$TERM, which actually terminates execution.

Routine P$TERM is a Pascal callable procedure without parameters. The following partial listing of P$TERM illustrates how it accesses the termination code:

```
        IDT   'P$TERM'
LO      TEXT  'P$TERM'
        DATA  2
        REF   ENT$S
        REF   S$TERM
        REF   T$CC,T$EC,T$VT,T$ES,T$MN
        DATA  EPI,LO
        DEF   P$TERM
P$TERM  BL    @ENT$S                    R12:=PROCESS RECORD ADDRESS
        DATA  40
*
*           ---TEST FOR NORMAL OR ABNORMAL COMPLETION---
*
        MOV   @>3A(R12),R8              R8:=TASK INFO.BLOCK ADDRESS
        MOV   @T$CC(R8),R1              R1:=COMPLETION CODE FROM P$UC
        MOV   @T$EC(R8),R3              R3:=ERROR CODE
        JEQ   L1                        SKIP IF NORMAL TERMINATION
        LI    R1,>C000                  SET $$CC TO INDICATE ABORT
L1      EQU   $
*
*           ---SET $$CC AND TERMINATE ---
*
        MOV   @T$VT(R8),R2              VARIABLE TEXT FOR MESSAGE
        MOV   @T$ES(R8),R3              ERROR SOURCE FLAGS
        MOV   @T$MN(R8),R4              MESSAGE NUMBER
EPI     BLWP  @S$TERM                   TERMINATE
        END
```

Routine S$TERM, which P$TERM calls with the value of $$CC in R1, sets the completion code synonym $$CC to the value in R1, terminates the task, and then displays a message specified by R2, R3, and R4. Although P$TERM has a regular Pascal entry point, it is not always possible for it to call Pascal routines since some errors do not leave a valid stack.

11.7.2 LUNO I/O Termination Routines

When a TIP task that is linked for LUNO I/O terminates, control passes to library routine TERM$, as described in the preceding paragraph. The TERM$ routine calls P$TERM to terminate execution. The version of P$TERM in library .TIP.LUNOBJ is simpler than the version in library .TIP.OBJ. It executes only an End-of-Task SVC.

11.7.3 Minimal Run-Time Termination Routines

When a TIP task that is linked for minimal run time terminates, control passes to library routine TERM$$. TERM$$ is an assembly language routine that is entered with the address of the process record for the terminating task in R12. The process record contains the WP, PC and ST values at the time of the error. The contents of these fields are not significant for a normal termination. The process record contains the termination code in bytes >3C and >3D.

After writing the termination message, TERM$$ calls P$STOP to terminate the task. The following is the listing of P$STOP:

```
        IDT   'P$STOP'
        DEF   P$STOP
P$STOP  LI    R1,>400     SET OP CODE FOR END-OF-TASK CALL
        XOP   R1,15       SUPERVISOR CALL
        END
```

If you replace TERM$$ and do not use the MESSAGE procedure, define a dummy message routine (as shown below) so that the message routine will not be included in the link.

```
        DEF   MESAG$
MESAG$  IDLE
```

12

Interfacing to Productivity Tools

12.1 GENERAL

This section describes productivity tools that can interface with TIP programs to perform several useful functions. These tools are optional software packages and are not automatically included with the operating system. The tools and their functions are as follows:

- TIFORM — Provides specialized software for control of video display terminal I/O formats. Consult the *TIFORM User's Guide* for details on using TIFORM.
- DBMS-990 — (Data Base Management System) — Enables definition and manipulation of a data base. Consult the *DNOS Data Base Management System Programmer's Guide* and the *DNOS Data Base Administrator User's Guide* for details of the definition and use of DBMS-990.
- Query-990 — Provides simplified data retrieval from a data base. Consult the *Query-990 User's Guide* for details on using Query-990.
- Sort/Merge — Controls file sorting and merging operations according to your criteria. Consult the *DNOS Sort/Merge User's Guide* for details on using Sort/Merge.

12.1.1 TIFORM

TIFORM is a software package that controls the I/O formats of a VDT. When using TIFORM, you can separate the application's procedural code from the characteristics of the terminal to be used. The screen formats, their components, and their attributes are called a *form*. A form can consist of either one formatted screen or several screens that are logically related. A form can be redesigned or installed on a different terminal without affecting the application. The application is also relieved of handling simple input edit errors. Generally, interactions with the terminal are shifted from the application program to TIFORM, leaving the application more flexible, transportable, and free to process data.

The interface routines for the TIFORM application program provide access to the form executor. Through these routines, the application issues commands and passes data to the executor and receives status and data in return. These routines handle all intertask communications between the application and the executor, providing a consistent error-correcting interface. All access to the application interface is through TIP procedure call statements.

The *TIFORM Reference Manual* contains information regarding forms creation and execution. The following paragraphs present examples of the calling sequence used in TIP programs and the linking requirements.

12.1.1.1 TIFORM Interface. The TIFORM interface package for TIP requires beginning addresses and sizes of variable length buffers. Entry points to the TIFORM interface are of the form, PX$aaa, where aaa defines the unique entry point. Table 12-1 lists the entry points of the package. Refer to the *TIFORM User's Guide* for an explanation of these calls.

Table 12-1. TIP Entry Points to TIFORM Routines

Calls	Meaning
PX$STS	Declare status block
PX$OF	Open form
PX$PS	Prepare segment
PX$WRI	Write a group
PX$REA	Read a group
PX$WWR	Write with reply
PX$WX	Write, indexed
PX$REX	Read, indexed
PX$RXC	Read, indexed, with cursor return
PX$WXR	Write, indexed, with reply
PX$WRC	Write, indexed, with reply and cursor return
PX$WM	Write message
PX$AEK	Arm event keys
PX$DAK	Disarm event keys
PX$CN	Control functions
PX$RF	Reset form
PX$RFX	Reset form indexed
PX$PKY	Print key command
PX$CHF	Change form
PX$ASN	Execute asynchronously
PX$SYN	Synchronize
PX$CIC	Change intertask communication
PX$CF	Close form

Each entry point must be declared as EXTERNAL. Each parameter of a declared entry point must be declared as a variable parameter (VAR), with the exception of all six-character names (e.g., form names, group names) and parameters that represent the size of data areas. These parameters must not be declared as variable. In addition, all data areas must be defined as dynamic arrays. The lower bound of all such arrays must be 1. For example:

```
VAR READ_DATA: PACKED ARRAY [1..?] OF CHAR;
```

It is recommended that the TIP upper bound function, UB(<array>) be used for determining the size of each array.

The file .S$TIFORM.PX$START contains a skeleton program which includes all of the type and procedure declarations for using TIFORM from TIP. It provides a convenient starting point for writing a program to use TIFORM.

Figure 12-1 is an example of a TIP program interfacing with TIFORM.

```
PROGRAM FORMS_DEMO;
TYPE
    C$2 = PACKED ARRAY [1..2] OF CHAR;
    C$6 = PACKED ARRAY [1..6] OF CHAR;
STATUS_BLOCK = PACKED RECORD
           FORM_STATUS: C$2;
           OPSYS_STATUS: C$2;
           EVENT_KEY: C$2;
           COMMAND: C$2;
           ITEM_NAME: C$6;
           INDEXES_CURSOR: PACKED ARRAY [1..12] OF CHAR;
           TEXT_LENGTH: INTEGER;
           FILLER_1: PACKED ARAY [1..12] OF CHAR;
END;

    DIR_NAME = PACKED ARRAY [1..10] OF CHAR;
      {DIR_NAME can vary from 2 to 48 bytes, depending
      on expected size of pathnames for the form
      program file.}

    TRM_NAME = PACKED ARRAY [1..4] OF CHAR;
      {TRM_NAME may vary from 2 to 8 bytes, depending on the
      expected size of the device name or synonym to be
      used for the terminal.}

    R_BUFF = PACKED ARRAY [1..48] OF CHAR;
      {Vary R_BUFF to accommodate maximum user input for
      any group read.}

VAR SBLOCK: STATUS_BLOCK;
    D_NAME: DIR_NAME; T_NAME: TRM_NAME;
    R_BUFFER: R_BUFF;

PROCEDURE PX$STS  (VAR SBLK: STATUS_BLOCK); EXTERNAL

PROCEDURE PX$OF   (FNAME: C$6;
                   VAR DIRNME: PACKED ARRAY [1..?] OF CHAR;
                       DIRSIZ: INTEGER;
                   VAR TRMNME: PACKED ARRAY [1..?] OF CHAR;
                       TRMSIZ: INTEGER;)
                   EXTERNAL;
```

Figure 12-1. Interfacing TIP and TIFORM (Sheet 1 of 2)

```
PROCEDURE PX$PS    (SNAME: C$6);
                   EXTERNAL;

PROCEDURE PX$REA   (GNAME: C$6;
                    VAR RDDATA:PACKED ARRAY [1..?] OF CHAR;
                        RBFSIZ:INTEGER);
                   EXTERNAL;

PROCEDURE PX$CF;
                   EXTERNAL;

BEGIN
          PX$STS (SBLOCK);     {Declare status block}

          D_NAME[1] :='';      {Use default synonym DIRECTORY
                                for form file

          T_NAME :='TERM';     {Use synonym TERM for terminal}

          PX$OF   ('FORM01', D_NAME, UB(D_NAME), T_NAME,
                   UB(T=NA);  {Open the form}

          PX$PS ('SEG001');    {Literal segment name is used}

                               {Activate the segment }

          PX$REA ('GROUP1', R_BUFFER, UB(R_BUFFER));  {Read user input}

                              {Process user input }

          PX$CF      {Close the form and terminate the executor}
END.
```

Figure 12-1. Interfacing TIP and TIFORM (Sheet 2 of 2)

12.1.1.2. Linking TIP and TIFORM. A TIP application program uses TIFORM by calling the routines listed in TABLE 12-1. These routines reside in the TIFORM/TIP interface module .S$TIFORM.O.PX$MTASK, which must be linked with the application program. Figure 12-2 is an example of a link control file for including TIFORM in a TIP program. Since TIFORM uses SCI, the libraries .TIP.LUNOBJ and .TIP.MINOBJ should not be used.

```
FORMAT IMAGE,REPLACE
LIBRARY .TIP.OBJ
TASK <task name>
INCLUDE (MAIN)
INCLUDE <user program>
INCLUDE .S$TIFORM.O.PX$MTASK      (includes PX$aaa routines)
END
```

Figure 12-2. Linking TIP and TIFORM

12.1.2 Data Base Management System (DBMS-990).

DBMS-990 provides a mechanism for organizing, storing, updating, and retrieving data by using mass-storage devices. Data base elements consist of data hierarchy and keys. The data hierarchy contains the logical data elements, and the keys allow random access to the data. Access to data elements is by logical names and data line types, so you need not know the exact organization of a file to manipulate the file contents. The DBMS-990 data hierarchy is oriented toward business documents such as invoices, purchase orders, and sales orders. A document is the basic means of initiating, executing, and recording business transactions.

DBMS-990 includes a data definition language (DDL) as a means of defining data elements within the data base. A data manipulation language (DML) is also included, enabling you to read, replace, add, and delete data. The *DNOS Data Base Management System Programmer's Guide* and the *DNOS Data Base Administrator User's Guide* describe the DBMS-990 system in detail. The following provides examples of using DBMS-990 with TIP.

12.1.2.1 Call Techniques to DBMS-990. When DBMS-990 security is installed, the appropriate file password must be provided. The password can be hard coded into the application program, solicited from a user, or obtained through an input parameter. When file-access checking is installed in DBMS, files must be opened and the appropriate file access specified before any data manipulation functions can be executed. The control block and line list parameters must be initialized before calling DBMS-990.

Figure 12-3 illustrates a TIP program interfacing with DBMS-990.

```
PROGRAM PEXMPL;
(*****************************************************************
 THIS PROGRAM WILL EXTRACT DATA FROM THE DBMS SALES ORDER FILE
 ABOUT SPECIFIC ITEM NUMBERS READ FROM A SEQUENTIAL INPUT TRANS-
 ACTION FILE. DATA RETRIEVED IS OUTPUT TO A SEQUENTIAL FILE
 THAT CAN BE DISPLAYED AFTER THE PROGRAM TERMINATES.

 THE ITEM DESCRIPTION AND UNIT PRICE ARE OBTAINED FROM THE ITEM
 FILE. THE SALES-ORDER NUMBER, SHIP-TO CUSTOMER NUMBER AND
 QUANTITY ON-ORDER FOR EACH SALES ORDER THAT CONTAINS THE ITEM
 ARE OBTAINED FROM THE SOFL FILE. THE SHIP-TO CUSTOMER NAME IS
 RETRIEVED FROM THE CUST FILE. ERROR MESSAGES ARE PRINTED WHERE
 APPLICABLE. THE FOLLOWING SYNONYMS MUST BE DEFINED:
     PINP  = PASCAL INPUT FILE ACCESS NAME
     POUT  = PASCAL OUTPUT FILE ACCESS NAME
*****************************************************************)
CONST
    EM1 = 'ERROR IN ';
    EM2 = ' FILE OPEN, STATUS=';
TYPE                                    (* DEFINE DATA TYPES *)
    C2      = PACKED ARRAY [1..2] OF CHAR;
    C4      = PACKED ARRAY [1..4] OF CHAR;
    C6      = PACKED ARRAY [1..6] OF CHAR;
    C20     = PACKED ARRAY [1..20] OF CHAR;
    DA_TYPE = (SOF2, SOF3, CUST, ITEM);
DATAREA = RECORD                        (* DEFINE RECORD AREAS *)
    CASE DA_TYPE OF
       SOF2 : (SHIP : C6);
       SOF3 : (QUAN : C4;  SONM : C6);
       CUST : (NAME : C20);
       ITEM : (DESC : C20; UPRC : C6);
END;
LINELIST = RECORD
    LL   : PACKED ARRAY [1..24] OF CHAR;
    TERM : INTEGER;
END;
CONTROLBLOCK = RECORD
    PSWD   : C4;
    FUNC   : C2;
    STAT   : C2;
    DBFILE : C4;
    LOC1   : C4;
    LOC2   : C4;
    KEYN   : C4;
    KEYV   : C6;
    TERM   : INTEGER;
END;
```

Figure 12-3. Interfacing TIP and DBMS-990 (Sheet 1 of 6)

```
(* VARIABLE DEFINITIONS *)
VAR
    ERR                 : BOOLEAN;
    ITEMNO              : C6;
    SOFLPK_LL           : LINELIST;
    SOFLSK_LL           : LINELIST;
    CUST_LL             : LINELIST;
    ITEM_LL             : LINELIST;
    DA                  : RECORD
       DATA : DATAREA;
       TERM : INTEGER;
       END;

    CB                  : CONTROLBLOCK;
    POUT                : TEXT;
    PINP                : TEXT;
    ITEM_EXISTS         : BOOLEAN;
    MORE_ITEMS          : BOOLEAN;
    SHIP_EXISTS         : BOOLEAN;
    ITEM_SOLD           : BOOLEAN;
    SLOC1               : C4;
    SLOC2               : C4;
    ODESCRPT            : C20;
    OPRICE              : C6;
    OQTYOO              : C4;
    OSONO               : C6;
    OSHIPNO             : C6;
    OSHIPNA             : C20;
(* DEFINE EXTERNAL PROCEDURE TO CALL DBMS *)
PROCEDURE DBMSYS (VAR CB : CONTROLBLOCK; VAR CBE : INTEGER;
                  VAR LL : LINELIST; VAR LLE : INTEGER;
                  VAR DA : DATAREA; VAR DAE : INTEGER);
                  EXTERNAL FORTRAN;
(*  ERROR ROUTINE *)
PROCEDURE ERR_ROUTINE;
BEGIN
    WRITELN (POUT, 'DBERROR STAT=', CB.STAT, ', DBFILE=', CB.DBFILE,
             ', KEYN=', CB.KEYN, ', KEYV=', CB.KEYV);
    END;
(*  SET LOC1 AND LOC2 TO "*" TO START *)
PROCEDURE INIT_LOCS;
BEGIN
    CB.LOC1 := '****';
    CB.LOC2 := '****';
    END;
```

Figure 12-3. Interfacing TIP and DBMS-990 (Sheet 2 of 6)

```
(* WRITE OUTPUT LINE *)
PROCEDURE WRITE_DATA;
BEGIN
    WRITELN(POUT, ITEMNO, ODESCRPT, '  ', OPRICE, '  ',
    OQTYOO, '  ', OSONO, '  ', OSHIPNO,  '  ' , OSHIPNA);
    END;
(* PROCEDURE TO GET CUSTOMER NAME FROM DBMS *)
(* CUSTOMER NAME IS RETRIEVED FROM THE CUSTOMER FILE "CUST" *)
PROCEDURE GET_NAME;
BEGIN
    CB.DBFILE := 'CUST';
    CB.KEYN := 'CUSN';
    CB.KEYV := DA.DATA.SHIP;
    INIT_LOCS;
    DBMSYS (CB, CB.TERM, CUST_LL, CUST_LL.TERM, DA.DATA, DA.TERM);
    IF CB.STAT = '**' AND CB.LOC1 <> '****'
       THEN OSHIPNA := DA.DATA.NAME
    ELSE
       IF CB.STAT = 'NK'
          THEN BEGIN
          OSHIPNA := '*** NO SHIP NAME    ';
          WRITE_DATA;
          END
       ELSE ERR_ROUTINE;
    END;
(*****   PROCEDURE TO GET THE SHIP-TO NAME FROM THE DBMS     *****)
(*                SHIP-TO NUMBER IS LINE=02 OF SOFL FILE          *)
PROCEDURE GET_SHIP;
BEGIN
    CB.DBFILE := 'SOFL';
    CB.KEYN := 'SONM';
    CB.KEYV := DA.DATA.SONM;
    INIT_LOCS;
    DA.DATA.SHIP := '      ';
    DBMSYS (CB, CB.TERM, SOFLPK_LL, SOFLPK_LL.TERM, DA.DATA, DA.TERM);
    IF CB.STAT = '**' THEN
        IF CB.LOC1 <> '****'
           THEN BEGIN
           SHIP_EXISTS := TRUE;
           OSHIPNO := DA.DATA.SHIP;
           END
        ELSE BEGIN
           SHIP_EXISTS := FALSE;
           OSHIPNA := '***NO SHIP IN SOFL  ';
           WRITE_DATA;
           END
    ELSE ERR_ROUTINE;
    END;
```

Figure12-3. Interfacing TIP and DBMS-990 (Sheet 3 of 6)

```
(*  GET SECONDARY KEY ITEM NUMBER FROM SALES ORDER FILE  *)
PROCEDURE GET_SOFL;
BEGIN
    CB.DBFILE := 'SOFL';
    CB.KEYN := 'ITEM';
    CB.KEYV := ITEMNO;
    IF MORE_ITEMS
       THEN BEGIN
       CB.LOC1 := SLOC1;
       CB.LOC2 := SLOC2;
       END
    ELSE INIT_LOCS;
    DBMSYS (CB, CB.TERM, SOFLSK_LL, SOFLSK_LL.TERM, DA.DATA, DA.TERM);
    IF CB.STAT ='**' THEN
       IF CB.LOC1 = '****'
          THEN MORE_ITEMS := FALSE
          ELSE BEGIN
             OQTYOO := DA.DATA.QUAN;
             OSONO  := DA.DATA.SONM;
             ITEM_SOLD := TRUE;
             SLOC1 := CB.LOC1;
             SLOC2 := CB.LOC2;
             END
    ELSE BEGIN
       MORE_ITEMS := FALSE;
       IF CB.STAT = 'NK'
          THEN  BEGIN
          OSHIPNA := '***ITEM NOT SOLD    ';
          WRITE_DATA;
          END
       ELSE ERR_ROUTINE;
       END;
    END;

(*                GET ITEM DESCRIPTION FROM ITEM FILE          *)
PROCEDURE GET_ITEM;
BEGIN
    CB.DBFILE := 'ITEM';
    CB.KEYN := 'ITMN';
    CB.KEYV := ITEMNO;
    INIT_LOCS;
    DBMSYS (CB, CB.TERM, ITEM_LL, ITEM_LL.TERM, DA.DATA, DA.TERM);
    IF CB.STAT = '**' AND CB.LOC1 <> '****'
       THEN BEGIN
       ODESCRPT := DA.DATA.DESC;
       OPRICE := DA.DATA.UPRC;
       ITEM_EXISTS := TRUE;
       END
```

Figure 12-3. Interfacing TIP and DBMS-990 (Sheet 4 of 6)

```
    ELSE
       IF CB.STAT = 'NK'
          THEN BEGIN
          ODESCRPT := '*ITEM DOES NOT EXIST';
          WRITE_DATA;
          END
       ELSE ERR_ROUTINE;
    END;
PROCEDURE PROC_ITEM;
BEGIN { PROCESSES EACH ITEM TO SEE IF A SHIP-TO CUSTOMER EXISTS }
    GET_SHIP;
    IF SHIP_EXISTS THEN GET_NAME;
    IF CB.STAT = '**' AND SHIP_EXISTS THEN WRITE_DATA;
    MORE_ITEMS := TRUE;
    IF SLOC2 = '****'
       THEN MORE_ITEMS := FALSE
       ELSE GET_SOFL;
    END;
(*****  GET ITEMS FROM "ITEM" FILE AND PROCESS THE SECONDARY *****
 *****              KEY ON THE SALES ORDER FILE.              *****)
PROCEDURE DBMS_ROUTINES;
BEGIN
    ITEM_EXISTS := FALSE;
    MORE_ITEMS := FALSE;
    ITEM_SOLD := FALSE;
    GET_ITEM;
    GET_SOFL;
    IF ITEM_EXISTS AND ITEM_SOLD
       THEN REPEAT PROC_ITEM UNTIL NOT MORE_ITEMS;
    END;
(*  PROCEDURE TO OPEN THE DATABASE FILES FOR FILE ACCESS CHECKING  *)
PROCEDURE OPEN_DATABASE_FILE(X:C4; VAR E:BOOLEAN);
BEGIN
    CB.DBFILE:=X;
    DBMSYS(CB,CB.TERM, CUST_LL,CUST_LL.TERM, DA.DATA,DA.TERM);
    IF CB.STAT <> '**'
       THEN BEGIN
       E:=TRUE;
       WRITELN(OUTPUT, EM1, X, EM2, CB.STAT);
       RESET(INPUT);
       END
    ELSE E:=FALSE;
    END;
```

Figure 12-3. Interfacing TIP and DBMS-990 (Sheet 5 of 6)

```
PROCEDURE INITIALIZATION;
BEGIN   (* INITIALIZE REMAINING AREAS AND PROCESS DATA *)
    SOFLPK_LL.LL := 'LINE=02*SHIP****RLSE    ';
    SOFLSK_LL.LL := 'LINE=03*QUANSONM****RLSE';
    CUST_LL.LL := 'LINE=01*NAME****RLSE    ';
    ITEM_LL.LL := 'LINE=01*DESCUPRC****RLSE';
    CB.PSWD := 'TEST';
(*** INITIALIZE THE CONTROL BLOCK TO OPEN THE DATABASE FILES ***)
    CB.FUNC := 'OF';
    CB.KEYN := 'SHRD';
    OPEN_DATABASE_FILE('CUST',ERR);IF ERR THEN ESCAPE INITIALIZATION;
    OPEN_DATABASE_FILE('ITEM',ERR);IF ERR THEN ESCAPE INITIALIZATION;
    OPEN_DATABASE_FILE('SOFL',ERR);IF ERR THEN ESCAPE INITIALIZATION;
    CB.FUNC := 'RF';
    CB.KEYN := 'SOMN';
    INIT_LOCS;
    END;
BEGIN   (* MAIN PROCESSING CYCLE *)
    RESET(PINP);
    REWRITE(POUT);
    INITIALIZATION;
    IF NOT ERR
       THEN REPEAT BEGIN
          READLN(PINP, ITEMNO);
          INIT_LOCS;
          ODESCRPT := '                    ';
          OPRICE := '      ';
          OQTYOO := '    ';
          OSONO := '      ';
          OSHIPNO := '       ';
          OSHIPNA := '                   ';
          DBMS_ROUTINES;
          WRITELN(POUT);
          END;
       UNTIL EOF(PINP);
    CLOSE (PINP);
    CLOSE (POUT);
END.
```

Figure 12-3. Interfacing TIP and DBMS-990 (Sheet 6 of 6)

12.1.2.2 Linking TIP and DBMS-990. Figure 12-4 shows an example of linking a TIP program to use DBMS-990.

```
FORMAT IMAGE, REPLACE
LIBRARY .TIP.OBJ
TASK <task name>
INCLUDE (MAIN)
INCLUDE <user main program>
INCLUDE .DBLIB.SNDMSG
INCLUDE .DBLIB.FRGMY
INCLUDE <user subroutines>
END
```

Figure 12-4. Linking TIP and DBMS-990

The files .DBLIB.SNDMSG and .DBLIB.FRGMY contain DBMS-990 utilities that are required for the TIP interface. They should be linked in with the task.

12.1.3 Query-990

The Query-990 software package provides a means of retrieving data from a DBMS-990 data base. Query-990 enables you to gather data and format a report without writing a program, or to interface with Query-990 from application programs.

The Query-990 processor produces a report or data file by accepting and executing a Query-990 language statement. An application program can execute a predefined statement that is stored on a file, or you can build, compile, and execute the Query-990 statement within the application program. Besides executing from an application program, you can invoke Query-990 interactively or in batch mode and pass to it a statement file created with the Query-990 editor or the system Text Editor. You can also build a Query-990 language statement by executing the Guided Query utility, which constructs the statement by prompting you with questions to determine the content and format of the report.

The Query-990 language is an English-like nonprocedural language with statements composed of several clauses. The clauses allow you to specify the content and format of each line, as well as the conditions that a data base record or line must meet to be qualified for output. Totals, counts, or averages can be performed on output fields and displayed in either default column headings or user-defined headings.

Using Query-990, you can specify a complex report in a few lines, while an application program to obtain the same report might take several hundred lines.

Figure 12-5 is an example of a TIP program that uses Query-990.

```
PROGRAM PEXPL;

(* PEXPL provides examples of the external definitions needed
  in a TIP program to access the Query subroutines. PEXPL
  also illustrates calls to the following subroutines:

  QINIT (initializes a Query processor)
  QEXEC (executes the Query object, lists the results to a file)
  QRECV (performs the function of QEXEC, but one line at a time)
  QEND  (deactivates a Query processor)                          *)

TYPE
  C80 = PACKED ARRAY[1..80] OF CHAR;
  C40 = PACKED ARRAY[1..40] OF CHAR;
  C16 = PACKED ARRAY[1..16] OF CHAR;
  C4  = PACKED ARRAY[1..4 ] OF CHAR;
  C2  = PACKED ARRAY[1..2 ] OF CHAR;

VAR
  QUERY_NUMBER,R_STATUS,EXTEND,LNG : INTEGER;
  PATHNM : C40;
  R_CODE : C2;
  PASSWORD : C4;
  DBUFF : C80;

(* The external definitions needed to call all of the interface
   routines are as follows:                                      *)

PROCEDURE QCOMP(VAR QUERY_NUMBER : INTEGER;
                VAR RETURN_STATUS : INTEGER;
                VAR RETURN_CODE : C2;
                VAR QUERY_STATEMENT : C2;
                VAR STATEMENT_LENGTH : INTEGER;
                VAR PASSWORD : C4;
                VAR FORMAT   : INTEGER;
                VAR LIST_TEXT : INTEGER); EXTERNAL;
(* NOTE-QCOMP is not used in this example                        *)

PROCEDURE QINIT(VAR QUERY_NUMBER : INTEGER;
                VAR RETURN_NUMBER : INTEGER;
                VAR RETURN_CODE : C2;
                VAR PATHNM : C2;
                VAR PASSWORD : C4); EXTERNAL;
```

Figure 12-5. Interfacing TIP and Query-990 (Sheet 1 of 3)

```
PROCEDURE QEXEC(VAR QUERY_NUMBER : INTEGER;
                VAR RETURN_STATUS : INTEGER;
                VAR RETURN_CODE : C2;
                VAR OUTPUT_PATHNAME : C2;
                VAR EXTEND : INTEGER); EXTERNAL;

PROCEDURE QCLR(VAR QUERY_NUMBER : INTEGER;
               VAR RETURN_STATUS : INTEGER); EXTERNAL;
(* NOTE-QCLR is not used in this example                          *)

PROCEDURE QEND(VAR QUERY_NUMBER : INTEGER;
               VAR RETURN_STATUS : INTEGER); EXTERNAL;

PROCEDURE QRECV(VAR QUERY_NUMBER : INTEGER;
                VAR RETURN_STATUS : INTEGER;
                VAR RETURN_CODE : C2;
                VAR DATA_BUFFER : C2;
                VAR BUFFER_LENGTH : INTEGER); EXTERNAL;

PROCEDURE QSEND(VAR QUERY_NUMBER : INTEGER;
                VAR RETURN_STATUS : INTEGER;
                VAR RETURN_CODE  : C2;
                VAR DATA_BUFFER : C2;
                VAR BUFFER_LENGTH : INTEGER); EXTERNAL;
(*NOTE-QSEND is not used in this example                         *)

BEGIN(*PEXPL*)
 REWRITE(OUTPUT);
 PASSWORD := 'DBMS';

(*Assume that prior to the execution of this program the desired
  Query has been compiled using QCOMP, and further assume that
  the resulting object resides on 'X.TEST.QUERYOBJ'.

  Assign the object pathname, choose a number to associate with a
  Query processor and initialize that processor                  *)

 PATHNM::C16 := 'X.TEST.QUERYOBJ ';(*NOTE-Trailing blank required*)
 QUERY_NUMBER := 2;
 QINIT(QUERY_NUMBER,R_STATUS,R_CODE,PATHNM::C2,PASSWORD);
 WRITELN('RETURN STATUS FROM QINIT = ',R_STATUS);

(* Check for normal completion of the initialization, specify a
  pathname for the result of the Query execution, execute the
  Query and end the Query processor                              *)
```

Figure 12-5. Interfacing TIP and Query-990 (Sheet 2 of 3)

```
 IF R_STATUS = 0 THEN
  BEGIN
    PATHNM::C16 := 'X.TEST.QUERYLST '; (*NOTE-Trailing blank *)
    EXTEND := 0;
    QEXEC(QUERY_NUMBER,R_STATUS,R_CODE,PATHNM::C2,EXTEND);
    WRITELN('RETURN STATUS FROM QEXEC = ',R_STATUS);
    QEND(QUERY_NUMBER,R_STATUS);
    WRITELN('RETURN STATUS FROM QEND = ',R_STATUS);
  END;

(*This section demonstrates a call to QRECV. Note that QEND has
  disassociated the Query processor and the assigned Query number,
  so this association must be reset. The pathname must be reset
  to the object file.                                          *)

 QUERY_NUMBER := 2;
 PATHNM::C16 := 'X.TEST.QUERYOBJ ';  (*NOTE-Trailing blank*)
 QINIT(QUERY_NUMBER,R_STATUS,R_CODE,PATHNM::C2,PASSWORD);
 WRITELN('RETURN STATUS FROM QINIT = ',R_STATUS);

(*Make repeated calls to QRECV until there are no more output lines*)

 IF R_STATUS = 0 THEN
  BEGIN
    REPEAT
     LNG := 80;
     QRECV(QUERY_NUMBER,R_STATUS,R_CODE,DBUFF::C2,LNG);
     IF LNG <> 0 AND R_CODE = '**' THEN
      WRITELN(DBUFF);
    UNTIL LNG = 0 OR R_CODE <> '**';

    WRITELN('RETURN STATUS FROM QRECV = ',R_STATUS);
    WRITELN('RETURN CODE FROM QRECV = ',R_CODE);

    QEND(QUERY_NUMBER,R_STATUS);
    WRITELN('RETURN STATUS FROM QEND = ',R_STATUS);
  END;
END.
```

Figure 12-5. Interfacing TIP and Query-990 (Sheet 3 of 3)

12.1.4 Linking TIP and Query-990

Figure 12-6 shows an example control file for linking a TIP program to use Query-990.

```
FORMAT IMAGE,REPLACE
LIBRARY .SCI990.S$OBJECT
LIBRARY .TIP.OBJ
TASK <task name>
INCLUDE (MAIN)
INCLUDE <user task pathname>
INCLUDE .QUERYLIB.PSCINT
INCLUDE .QUERYLIB.PLIOBJ
END
```

Figure 12-6. Linking TIP and Query-990

12.1.5 Sort/Merge

The Sort/Merge utility is used to sort one file or to merge up to five sorted files into a single file, according to user-specified criteria. The Sort/Merge task can be executed as a utility or from a user program. When executed as a utility, Sort/Merge processes user-entered control statements and handles its own I/O. SCI provides commands to access Sort/Merge in batch or interactive mode. When executing Sort/Merge from a TIP program, the procedure call statement is used, and control statements and/or data records are passed between the calling task and Sort/Merge.

In a sort or merge operation, the Sort/Merge program does not alter the contents of the input files. Instead, sequenced records are written to a separate output file. To perform a sort, Sort/Merge uses an intermediate file called a *work file*.

Sort/Merge is accessible in the following ways:

- Through SCI, which interprets interactive and batch commands, executes Sort/Merge, and passes control statements to it.
- Through tasks that execute Sort/Merge and pass control statements it. The calling task can send records to Sort/Merge and/or receive the processed records returned by Sort/Merge.
- By executing the Sort/Merge task and specifying the pathname of a file or physical device that contains the control statement specifications.

The different types of sorts and merges and the control statements necessary to specify the processing requirements are detailed in the *DNOS Sort/Merge User's Guide*.

12.1.5.1 TIP and Sort/Merge Example. The following examples illustrate TIP routines that call Sort/Merge. Figure 12-7 is a TIP routine that calls Sort/Merge to input records from a disk file and output the sorted records to a second disk file. This example calls the Sort/Merge routines SRTINT and SMSTAT. Figure 12-8 is a TIP routine that calls Sort/Merge and passes records read by the TIP routine to Sort/Merge. The sorted records are returned to the TIP routine, which writes them to a disk file. This example calls the Sort/Merge routines SRTINT, SENREC, and RCVREC.

```
PROGRAM PCNPNP;
  (*
    PROGRAM TITLE: PCNPNP

    ABSTRACT:
       THIS IS A SORT/MERGE INTERPROCESS COMMUNICATION
       TEST. IT TESTS THE CASE WHERE BOTH INPUT AND
       OUTPUT FILES ARE ACCESSED DIRECTLY BY SORT/MERGE.

    *)

CONST   MAX_NO_RECS = 10;
        SECNDS = 26;

TYPE    CONTROL_RECORD = PACKED ARRAY [1..44] OF CHAR;

VAR
   (* THE FOLLOWING STATEMENTS ARE THE CONTROL RECORDS FOR
      SRTINT. THEY MUST BE DECLARED AS ONE CONTIGUOUS BLOCK *)
        SCB_HEADER                  : CONTROL_RECORD;
        SCB_OUTFILE_SPEC            : CONTROL_RECORD;
        SCB_OUTFILE_SPEC2           : CONTROL_RECORD;
        SCB_WRKFILE_SPEC            : CONTROL_RECORD;
        SCB_INPFILE_SPEC            : CONTROL_RECORD;
        SCB_INPFILE_SPEC2           : CONTROL_RECORD;
        SCB_REF_DESC_1              : CONTROL_RECORD;
        SCB_REF_DESC_2              : CONTROL_RECORD;
        SCB_REF_DESC_3              : CONTROL_RECORD;
        SCB_ENDKRD                  : CONTROL_RECORD;
   (*   END OF SORT CONTROL RECORDS                  *)

        STATIS                      : INTEGER;

PROCEDURE SRTINT (VAR SORT_CONTROL_BLOCK : CONTROL_RECORD;
                  MAX_NO_RECS : INTEGER;
                  VAR STATIS : INTEGER); EXTERNAL FORTRAN;

PROCEDURE SENREC (VAR STATIS : INTEGER); EXTERNAL FORTRAN;
```

Figure 12-7. TIP Interfacing With Sort/Merge to Input and Output Files From Disk (Sheet 1 of 2)

```
BEGIN

   (* INITIALIZE SORT CONTROL BLOCK  *)

       SCB_HEADER :=
         '00000HSORTR     6D        4   80  12000     ';
       SCB_OUTFILE_SPEC :=
         '00001DOS.OUTX                               ';
       SCB_OUTFILE_SPEC2 :=
         '00002DA00800864                             ';
       SCB_WRKFILE_SPEC :=
         '00003DWEDS01                                ';
       SCB_INPFILE_SPEC :=
         '00004DIS.SRTDAT                             ';
       SCB_INPFILE_SPEC2 :=
         '00008DA0080         400                     ';
       SCB_REF_DESC_1 :=
         '00010FNC  32  37                            ';
       SCB_REF_DESC_2 :=
         '00014FDC   1  31                            ';
       SCB_REF_DESC_3 :=
         '00016FDC  38  80                            ';
       SCB_ENDKRD :=
         '/*                                          ';

  (*  INITIALIZE SORT/MERGE  *)

       SRTINT(SCB_HEADER,MAX_NO_RECS,STATIS);
       IF STATIS <> 0 THEN
          MESSAGE ( 'ERROR IN SRTINT CALL. ')

       ELSE
       BEGIN
  (*  WAIT FOR SORT/MERGE TO COMPLETE   *)
         SMSTAT(STATIS);
         IF STATIS <> 0 THEN
           MESSAGE ( 'ERROR IN SMSTAT CALL. ')

      END;
END.
```

Figure 12-7. TIP Interfacing With Sort/Merge to Input and Output Files From Disk (Sheet 2 of 2)

```
PROGRAM PCPP;
  (*
  PROGRAM TITLE: PCPP

  ABSTRACT:
    THIS IS A SORT/MERGE INTERPROCESS COMMUNICATION
    TEST. IT TESTS THE CASE WHERE BOTH INPUT AND
    OUTPUT ARE DIRECTED BY THE PASCAL PROGRAM (@PROC@).

  *)

CONST MAX_NO_RECS = 10;
      RECORD_LENGTH = 80;
      ALLDONE = 0;

TYPE  CONTROL_RECORD = PACKED ARRAY [1..44] OF CHAR;
      DATA_REC = PACKED ARRAY [1..80] OF CHAR;

VAR
  (* THE FOLLOWING STATEMENTS ARE THE CONTROL RECORDS FOR
    SRTINT. THEY MUST BE DECLARED AS ONE CONTIGUOUS BLOCK *)
      SCB_HEADER                  : CONTROL_RECORD;
      SCB_OUTFILE_SPEC            : CONTROL_RECORD;
      SCB_OUTFILE_SPEC2           : CONTROL_RECORD;
      SCB_WRKFILE_SPEC            : CONTROL_RECORD;
      SCB_INPFILE_SPEC            : CONTROL_RECORD;
      SCB_INPFILE_SPEC2           : CONTROL_RECORD;
      SCB_REF_DESC_1              : CONTROL_RECORD;
      SCB_REF_DESC_2              : CONTROL_RECORD;
      SCB_REF_DESC_3              : CONTROL_RECORD;
      SCB_ENDKRD                  : CONTROL_RECORD;
  (*  END OF SORT CONTROL RECORDS                  *)

      INFILREC                    : DATA_REC;
      OUTFILREC                   : DATA_REC;
      STATIS                      : INTEGER;
      BYTES_RECEIVED              : INTEGER;
      ESC_FLAG                    : BOOLEAN;
      INFILE                      : FILE OF DATA_REC;
      OUTFL                       : FILE OF DATA_REC;

PROCEDURE SRTINT (VAR SORT=CONTROL=BLOCK : CONTROL=RECORD;
                  MAX=NO=RECS : INTEGER;
                  VAR STATIS : INTEGER); EXTERNAL FORTRAN;
PROCEDURE RCVREC (VAR OUTFILREC : DATA_REC;
                  RECORD_LENGTH : INTEGER;
                  VAR STATIS : INTEGER ); EXTERNAL FORTRAN;
```

Figure 12-8. TIP Interfacing With Sort/Merge to Input and Output Files From Calling Task (Sheet 1 of 3)

```
PROCEDURE RCVREC (VAR OUTFILREC : DATA_REC;
                  RECORD_LENGTH : INTEGER;
                  VAR BYTES_RECEIVED : INTEGER;
                  VAR STATIS : INTEGER ); EXTERNAL FORTRAN;
BEGIN

   (* INITIALIZE SORT CONTROL BLOCK  *)

      SCB_HEADER :=
        '00000HSORTR     6A        4   80  12000     ';
      SCB_OUTFILE_SPEC :=
        '00001DOS@PROC@                              ';
      SCB_OUTFILE_SPEC2 :=
        '00002DA00800864                             ';

      SCB_WRKFILE_SPEC :=
        '00003DWEDS01                                ';
      SCB_INPFILE_SPEC :=
        '00004DIS@PROC@                              ';
      SCB_INPFILE_SPEC2 :=
        '00008DA0080         400                     ';
      SCB_REF_DESC_1 :=
        '00010FNC  32  37                            ';
      SCB_REF_DESC_2 :=
        '00014FDC   1  31                            ';
      SCB_REF_DESC_3 :=
        '00016FDC  38  80                            ';
      SCB_ENDKRD :=
        '/*                                          ';

      RESET(INFILE);

  (*  INITIALIZE SORT/MERGE  *)

      SRTINT(SCB_HEADER,MAX_NO_RECS,STATIS);
      IF STATIS <> 0 THEN
         MESSAGE ( 'ERROR IN SRTINT CALL. ')
      ELSE
      BEGIN
               VAR STATIS : INTEGER); EXTERNAL FORTRAN;
```

Figure 12-8. TIP Interfacing With Sort/Merge to Input and Output Files From Calling Task (Sheet 2 of 3)

```
(*  READ RECORDS FROM INPUT FILE AND SEND TO SORT/MERGE *)
      REPEAT
        READ(INFILE,INFILREC);
        SENREC(INFILREC,RECORD_LENGTH,STATIS);
      UNTIL EOF(INFILE) OR (STATIS <> 0);

      IF STATIS <> 0 THEN
        MESSAGE ( 'ERROR IN SENREC CALL. ')
      ELSE
      BEGIN
        (* SEND RECORD LENGTH OF ZERO  *)
        SENREC(INFILREC,ALLDONE,STATIS);
        IF STATIS <> 0 THEN
          MESSAGE ( 'ERROR IN SENREC CALL. ')
        ELSE
        BEGIN
         (* OPEN OUTPUT FILE  *)
          REWRITE (OUTFL);
          ESC_FLAG := FALSE;
          WHILE ESC_FLAG = FALSE DO
          BEGIN
(*  WRITE SORTED RECORDS TO OUTPUT FILE  *)
           RCVREC(OUTFILREC,RECORD_LENGTH,
                  BYTES_RECEIVED,STATIS);
           IF (STATIS = 0) AND (BYTES_RECEIVED <> 0 )
              THEN WRITE (OUTFL,OUTFILREC)
              ELSE ESC_FLAG := TRUE
          END;
          IF STATIS = 1 THEN
           MESSAGE ( 'ERROR IN RCVREC CALL. ')
        END
      END
    END;
END.
```

Figure 12-8. TIP Interfacing With Sort/Merge to Input and Output Files From Calling Task (Sheet 3 of 3)

12.1.5.2 Linking TIP and Sort/Merge. The library SSMRG.SMLIB contains the following Sort/Merge interface routines: SRTINT, SENREC, RCVREC, SMSTAT. In the linking example that follows, all of these subroutines are included. A TIP program requires only PSCINT, SRTINT and other subroutines that are actually called. PSCINT and SRTINT must be either in the root segment in overlay structures or in the same segment as SENREC, RCVREC, and SMSTAT. SENREC, RCVREC, and SMSTAT call internal subroutines contained in SRTINT. PSCINT is called by all of the other subroutines. Figure 12-9 shows a partial control file for linking TIP and Sort/Merge.

```
FORMAT IMAGE,REPLACE
LIBRARY .TIP.OBJ
PHASE 0, <task name>
INCLUDE (MAIN)
INCLUDE <user main program>
INCLUDE S$SMRG.SMLIB.SRTINT
INCLUDE S$SMRG.SMLIB.PSCINT
INCLUDE S$SMRG.SMLIB.SENREC
INCLUDE S$SMRG.SMLIB.RCVREC
INCLUDE S$SMRG.SMLIB.SMSTAT
INCLUDE <user subroutines>
END
```

Figure 12-9. Linking TIP and Sort/Merge

13

Reverse Assembler (RASS) Utility Details

13.1 GENERAL

The Reverse Assembler (RASS) takes an object module written by the TIP compiler and provides a corresponding 990 assembly language source program. The output of RASS may be assembled directly. By submitting an object module to RASS and obtaining the assembly language source code, you can perform manual optimization when appropriate. The assembly language source code allows debugging at the machine-language level.

13.2 REQUIRED FILES

RASS requires the following files:

File	I/O	Contents
OBJECT	I	Object file to be processed
OUTPUT	O	Assembly language listing
SYSMSG	O	System messages

13.3 EXECUTING RASS

TIP software includes an SCI procedure XRASS for executing RASS. Enter XRASS at any time in response to the SCI prompt [ ]. The following information is requested:

```
EXECUTE REVERSE ASSEMBLER <VERSION: X.X.X YYDDD>
                   OBJECT:  pathname@
                  LISTING:  pathname@
                 MESSAGES:  pathname@
                     MODE:  {FOREGROUND/BACKGROUND} (FOREGROUND)
                   MEMORY:  integer,integer (2,8)
```

The first three items require access names of devices or files, as follows:

- OBJECT — The access name of the object file (input)
- LISTING — The access name of the device or file for the listing (both errors and assembly language source)
- MESSAGES — The access name of the device or file for the system messages

The MODE may be either FOREGROUND or BACKGROUND. The MEMORY item is an ordered pair specifying stack and heap for the execution of RASS.

13.4 EXAMPLE LISTING

Figure 13-1 is an example of the assembly listing produced by RASS. The program listed consists of three modules. The listing for each module begins with the IDT directive that contains the module name. This is followed by a comment line that provides a heading for the columns at the right.

```
*
      IDT  'CCHAR   '              12/23/83  11:11:40
*
*                                  DXPSCL        1.8.0
*
*                                    LC    HEX    CHAR
      PSEG
D0000 DATA >4343                   0000   4343   CC
      DATA >4841                   0002   4841   HA
      DATA >5220                   0004   5220   R
      DATA >2020                   0006   2020
      DATA >0002                   0008   0002   ..
D000A DATA >0030                   000A   0030   .0
D000C DATA >0039                   000C   0039   .9
      DATA L0072                   000E   0072+  ..
      DATA D0000                   0010   0000+  ..
*
      DEF  CCHAR
      REF  ENT$S
      REF  RET$S
*                                  LC   WORD(S)
CCHAR EQU  $
      BL   @ENT$S                  0012   06A0   0000
      DATA >003A                   0016   003A
      MOV  @>0034(R9),R5           0018   C169   0034
      CLR  *R5                     001C   04D5
      LI   R6,>0001                001E   0206   0001
      MOV  @>0036(R9),@>0038(R9)   0022   CA69   0036   0038
L0028 EQU  $
      C    R6,@>0038(R9)           0028   8A46   0038
      JGT  L0072                   002C   1522
      MOV  R6,R5                   002E   C146
      SLA  R5,1                    0030   0A15
      AI   R5,>0026                0032   0225   0026
      A    R9,R5                   0036   A149
      C    *R5,@D000A              0038   8815   000A+
      JL   L006E                   003C   1A18
      MOV  R6,R5                   003E   C146
      SLA  R5,1                    0040   0A15
      AI   R5,>0026                0042   0225   0026
      A    R9,R5                   0046   A149
      C    *R5,@D000C              0048   8815   000C+
      JH   L006E                   004C   1B10
```

Figure 13-1. RASS Listing Example (Sheet 1 of 5)

```
          MOV  @_0034(R9),R5          004E   C169   0034
          LI   R3,_000A               0052   0203   000A
          MPY  *R5,R3                 0056   38D5
          MOV  R6,R5                  0058   C146
          SLA  R5,1                   005A   0A15
          AI   R5,>0026               005C   0225   0026
          A    R9,R5                  0060   A149
          A    *R5,R4                 0062   A115
          AI   R4,>FFD0               0064   0224   FFD0
          MOV  @>0034(R9),R5          0068   C169   0034
          MOV  R4,*R5                 006C   C544
L006E     EQU  $
          INC  R6                     006E   0586
          JMP  L0028                  0070   10DB
L0072     EQU  $
          B    @RET$S                 0072   0460   0000
          END
*
          IDT  'CINT    '             12/23/83  11:11:45
*
*                                     DXPSCL   1.8.0
*
*                                       LC    HEX    CHAR
          PSEG
D0000     DATA >4349                  0000   4349   CI
          DATA >4E54                  0002   4E54   NT
          DATA >2020                  0004   2020
          DATA >2020                  0006   2020
          DATA >0002                  0008   0002   ..
          DATA L0058                  000A   0058+  .X
          DATA D0000                  000C   0000+  ..
*
          DEF  CINT
          REF  ENT$M
          REF  DIV$
          REF  CINT
          REF  PUTCH$
          REF  RET$M
*                                       LC    WORD(S)
CINT      EQU  $
          BL   @ENT$M                 000E   06A0   0000
          DATA >002C                  0012   002C
          MOV  @>0002(R12),R15        0014   C3EC   0002
          MOV  @>0028(R9),R8          0018   C229   0028
          LI   R12,>000A              001C   020C   000A
          BL   @DIV$                  0020   06A0   0000
          MOV  R7,@>002A(R9)          0024   CA47   002A
          MOV  @>002A(R9),R7          0028   C1E9   002A
          JEQ  L003A                  002C   1306
          MOV  @>002A(R9),@>0028(R10) 002E   CAA9   002A   0028
          LI   R11,CINT               0034   020B   0000
          BLWP R10                    0038   040A
```

Figure 13-1. RASS Listing Example (Sheet 2 of 5)

```
L003A  EQU   $
       MOV   @>0028(R9),R8          003A   C229   0028
       LI    R12,>000A              003E   020C   000A
       BL    @DIV$                  0042   06A0   0022
       MOV   R8,R6                  0046   C188
       AI    R6,>0030               0048   0226   0030
       SB    R6,R                   004C   7186
       MOV   @>0040(R15),R12        004E   C32F   0040
       MOV   R6,R7                  0052   C1C6
       BL    @PUTCH$                0054   06A0   0000
L0058  EQU   $
       B     @RET$M                 0058   0460   0000
       END
*
       IDT   'DIGIO   '             12/23/83   11:11:55
*
*                                   DXPSCL       1.8.0
*
*                                     LC     HEX   CHAR
       PSEG
D0000  DATA  >4449                  0000   4449   DI
       DATA  >4749                  0002   4749   GI
       DATA  >4F20                  0004   4F20   O
       DATA  >2020                  0006   2020
D0008  DATA  >0001                  0008   0001   ..
D000A  DATA  >0060                  000A   0060   ..
D000C  DATA  >494E                  000C   494E   IN
       DATA  >5055                  000E   5055   PU
       DATA  >5420                  0010   5420   T
       DATA  >2020                  0012   2020
D0014  DATA  D000C                  0014   000C+  ..
D0016  DATA  >0002                  0016   0002   ..
D0018  DATA  >0013                  0018   0013   ..
D001A  DATA  >454E                  001A   454E   EN
       DATA  >5445                  001C   5445   TE
       DATA  >5220                  001E   5220   R
       DATA  >3120                  0020   3120   1
       DATA  >544F                  0022   544F   TO
       DATA  >2035                  0024   2035   5
       DATA  >2044                  0026   2044   D
       DATA  >4947                  0028   4947   IG
       DATA  >4954                  002A   4954   IT
       DATA  >5320                  002C   5320   S
D002E  DATA  D001A                  002E   001A+  ..
D0030  DATA  >008E                  0030   008E   ..
D0032  DATA  >0019                  0032   0019   ..
       DATA  L0112                  0034   0112+  ..
       DATA  D0000                  0036   0000+  ..
*
       DEF   PSCL$$
       REF   ENT$1
       REF   MOV$4
       REF   FL$INI
```

Figure 13-1. RASS Listing Example (Sheet 3 of 5)

```
        REF   WRS$T
        REF   WRLN$
        REF   REST$T
        REF   EOLN$
        REF   GET$CH
        REF   MOV$6
        REF   CCHAR
        REF   CINT
        REF   CLS$O$
        REF   RET$1
*                                        LC    WORD(S)
PSCL$$ EQU   $
        BL    @ENT$1                     0038  06A0  0000
        DATA  >0090                      003C  0090
        MOV   R9,@>0028(R10)             003E  CA89  0028
        A     @D000A,@>0028(R10)         0042  AAA0  000A +    0028
        MOV   @D0014,R7                  0048  C1E0  0014 +
        MOV   R10,R8                     004C  C20A
        AI    R8,>002A                   004E  0228  002A
        BL    @MOV$4                     0052  06A0  0000
        MOV   @D0008,@>0032(R10)         0056  CAA0  0008 +    0032
        MOV   @D0016,@>0034(R10)         005C  CAA0  0016 +    0034
        LI    R11,FL>INI                 0062  020B  0000
        BLWP  R10                        0066  040A
        MOV   @>0040(R9),@>0028(R10)     0068  CAA9  0040      0028
        MOV   @D0018,@>002A(R10)         006E  CAA0  0018 +    002A
        MOV   @D0018,@>002C(R10)         0074  CAA0  0018 +    002C
        MOV   @D002E,@>002E(R10)         007A  CAA0  002E +    002E
        LI    R11,WRS$T                  0080  020B  0000
        BLWP  R10                        0084  040A
        MOV   @>0040(R9),@>0028(R10)     0086  CAA9  0040      0028
        LI    R11,WRLN$                  008C  020B  0000
        BLWP  R10                        0090  040A
        MOV   @>0060(R9),@>0028(R10)     0092  CAA9  0060      0028
        LI    R11,REST$T                 0098  020B  0000
        BLWP  R10                        009C  040A
        INC   @>008C(R9)                 009E  05A9  008C
LOOA2   EQU   $
        MOV   @>0060(R9),R12             00A2  C329  0060
        BL    @EOLN                      00A6  06A0  0000
        JGT   LOOCA                      00AA  150F
        MOV   @>008C(R9),R15             00AC  C3E9  008C
        SLA   R15,1                      00B0  0A1F
        AI    R15,>007E                  00B2  022F  007E
        A     R9,R15                     00B6  A3C9
        MOV   @>0060(R9),R12             00B8  C329  0060
        BL    @GET$CH                    00BC  06A0  0000
        SWPB  R7                         00C0  06C7
        MOV   R7,*R15                    00C2  C7C7
        INC   @>008C(R9)                 00C4  05A9  008C
        JMP   LOOA2                      00C8  10EC
LOOCA   EQU   $
        DEC   @>008C(R9)                 00CA  0629  008C
```

Figure 13-1. RASS Listing Example (Sheet 4 of 5)

```
        MOV   R9,R7                    00CE   C1C9
        AI    R7,>0080                 00D0   0227   0080
        MOV   R10,R8                   00D4   C20A
        AI    R8,>0028                 00D6   0228   0028
        BL    @MOV$6                   00DA   06A0   0000
        MOV   R9,@>0034(R10)           00DE   CA89   0034
        A     @D0030,@>0034(R10)       00E2   AAA0   0030+  0034
        MOV   @>008C(R9),@>0036(R10)   00E8   CAA9   008C   0036
        LI    R11,CCHAR                00EE   020B   0000
        BLWP  R10                      00F2   040A
        A     @D0032,@>008E(R9)        00F4   AA60   0032+  008E
        MOV   @>008E(R9),@>0028(R10)   00FA   CAA9   008E   0028
        LI    R11,CINT                 0100   020B   0000
        BLWP  R10                      0104   040A
        MOV   @>0040(R9),@>0028(R10)   0106   CAA9   0040   0028
        LI    R11,WRLN                 010C   020B   008E
        BLWP  R10                      0110   040A
LO112   EQU   $
        MOV   R9,@>0028(R10)           0112   CA89   0028
        A     @D000A,@>0028(R10)       0116   AAA0   000A+  0028
        LI    R11,CLS                  011C   020B   0000
        BLWP  R10                      0120   040A
        B     @RET$1                   0122   0460   0000
        END
```

Figure 13-1. RASS Listing Example (Sheet 5 of 5)

Following the heading are the PSEG directive, the DATA directive, the DEF directive, and the REF directive. The location counter value is shown for each data word, as are the contents, both in hexadecimal. When the data word represents a relocatable address, a plus sign (+) follows the hexadecimal value. The representation of the word as a pair of ASCII characters is shown at the end of each line. Each unprintable character is shown as a period (.).

Following the directives, another comment provides a heading for the instructions that follow. For each instruction, the listing shows the location counter value and the value of the word or words of the machine-language instruction in hexadecimal format. The file that contains the listing (OUTPUT) may be used as a source code file for the assembler. The assembler treats the values that appear at the right end of most lines as comments.

RASS is specifically designed to process the object files written by CODEGEN and has limited application for processing other object code files. It recognizes only the object output tag characters and operation codes supplied by CODEGEN. The directives are supplied in accordance with TIP format, since object code seldom contains enough information to explicitly identify the directives. When RASS processes object code that contains a tag character or operation code that it does not recognize, an error termination occurs. Even if the object code does not contain unrecognizable data, only those words that RASS correctly interprets as instructions are correctly listed.

The first eight bytes of each module contain the module name (program, procedure, or function name). The next two bytes contain the nesting level: 1 for the main program, 2 for procedures and routines declared in the main program, and so on. The TIP source program corresponding to the example in Figure 13-1 is shown in Figure 5-3. It consists of main program DIGIO, which declares procedures CCHAR and CINT. Therefore, the three modules in Figure 13-1 are CCHAR and CINT at level 2 and main program DIGIO at level 1.

Careful comparison of the source code in Figures 5-3 and 13-1 shows how the TIP compiler implements Pascal statements in assembly language. To perform manual optimization of the assembly language source code, omit any redundant source lines and reassemble the object module.

Appendix A

Keycap Cross-Reference

Generic keycap names that apply to all terminals are used for keys on keyboards throughout this manual. This appendix contains specific keyboard information to help you identify individual keys on any supported terminal. For instance, every terminal has an Attention key, but not all Attention keys look alike or have the same position on the keyboard. You can use the terminal information in this appendix to find the Attention key on any terminal.

The terminals supported are the 931 VDT, 911 VDT, 915 VDT, 940 EVT, the Business System terminal, and hard-copy terminals (including teleprinter devices). The 820 KSR has been used as a typical hard-copy terminal. The 915 VDT keyboard information is the same as that for the 911 VDT except where noted in the tables.

Appendix A contains three tables and keyboard drawings of the supported terminals.

Table A-1 lists the generic keycap names alphabetically and provides illustrations of the corresponding keycaps on each of the currently supported keyboards. When you need to press two keys to obtain a function, both keys are shown in the table. For example, on the 940 EVT the Attention key function is activated by pressing and holding down the Shift key while pressing the key labeled PREV FORM NEXT. Table A-1 shows the generic keycap name as Attention, and a corresponding illustration shows a key labeled SHIFT above a key named PREV FORM NEXT.

Function keys, such as F1, F2, and so on, are considered to be already generic and do not need further definition. However, a function key becomes generic when it does not appear on a certain keyboard but has an alternate key sequence. For that reason, the function keys are included in the table.

Multiple key sequences and simultaneous keystrokes can also be described in generic keycap names that are applicable to all terminals. For example, you use a multiple key sequence and simultaneous keystrokes with the log-on function. You log on by *pressing the Attention key, then holding down the Shift key while you press the exclamation (!) key.* The same information in a table appears as *Attention/(Shift)!*.

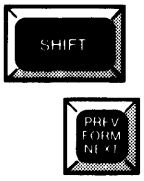
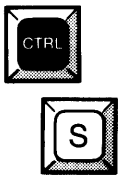
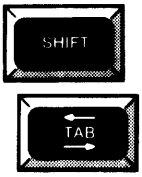
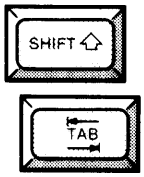
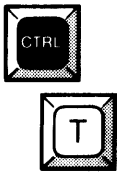
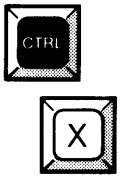
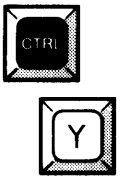
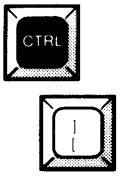
Table A-2 shows some frequently used multiple key sequences.

Table A-3 lists the generic names for 911 keycap designations used in previous manuals. You can use this table to translate existing documentation into generic keycap documentation.

Figures A-1 through A-5 show diagrams of the 911 VDT, 915 VDT, 940 EVT, 931 VDT, and Business System terminal, respectively. Figure A-6 shows a diagram of the 820 KSR.

2274834 (1/14)

Table A-1. Generic Keycap Names

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Alternate Mode	None				None
Attention[2]					
Back Tab	None			None	
Command[2]					
Control					
Delete Character					None
Enter					
Erase Field					

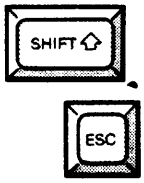
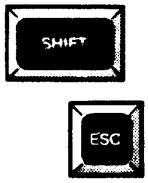
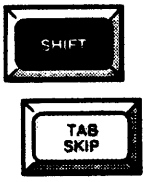
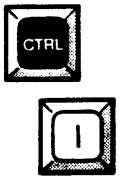
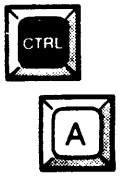
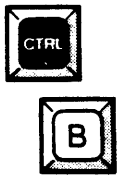
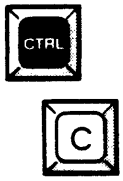
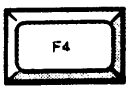
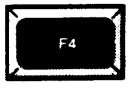
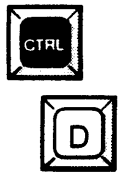
Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

[2]On a 915 VDT the Command Key has the label F9 and the Attention Key has the label F10.

2284734 (2/14)

Table A-1. Generic Keycap Names (Continued)

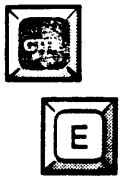
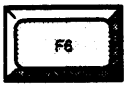
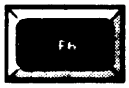
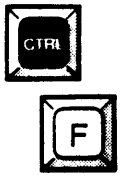
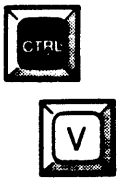
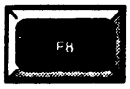
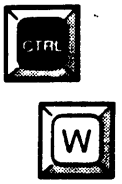
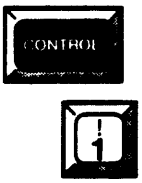
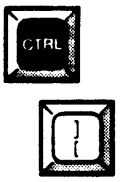
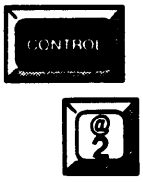
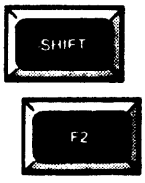
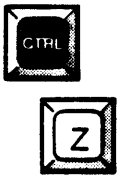
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Erase Input					
Exit					
Forward Tab					
F1					
F2					
F3					
F4					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (3/14)

Table A-1. Generic Keycap Names (Continued)

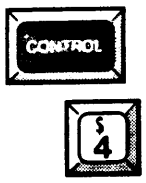
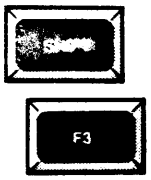
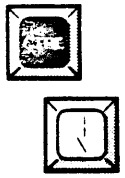
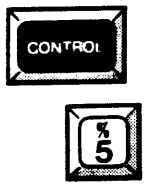
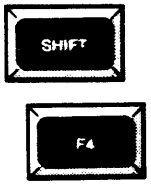
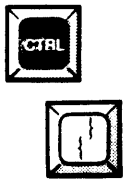
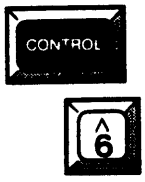
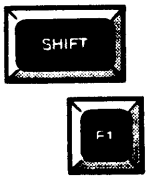
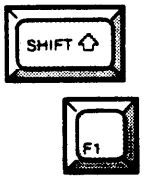
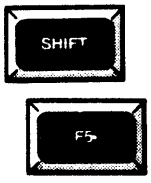
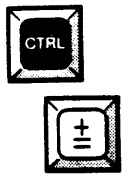
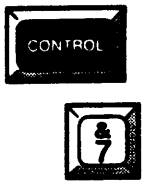
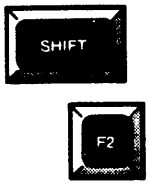
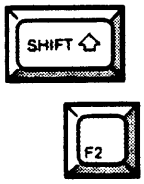
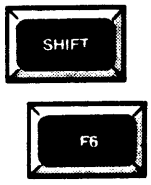
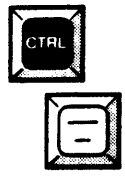
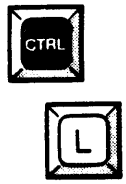
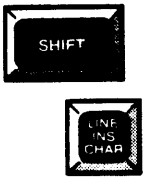
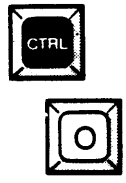
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
F5					
F6					
F7					
F8					
F9					
F10					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (4/14)

Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
F11					
F12					
F13					
F14					
Home					
Initialize Input					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions

2284734 (5/14)

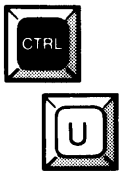
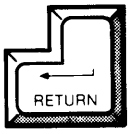
Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Insert Character					None
Next Character	 or  				None
Next Field	 		 	 	None
Next Line					  or 
Previous Character	 or 				None
Previous Field		 			None

Notes:
[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (6/14)

Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Previous Line					
Print					None
Repeat		See Note 3	See Note 3	See Note 3	None
Return					
Shift					
Skip					None
Uppercase Lock					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

[2]The keyboard is typamatic, and no repeat key is needed.

2284734 (7/14)

Table A-2. Frequently Used Key Sequences

Function	Key Sequence
Log-on	Attention/(Shift)!
Hard-break	Attention/(Control)x
Hold	Attention
Resume	Any key

Table A-3. 911 Keycap Name Equivalents

911 Phrase	Generic Name
Blank gray	Initialize Input
Blank orange	Attention
Down arrow	Next Line
Escape	Exit
Left arrow	Previous Character
Right arrow	Next Character
Up arrow	Previous Line

2284734 (8/14)

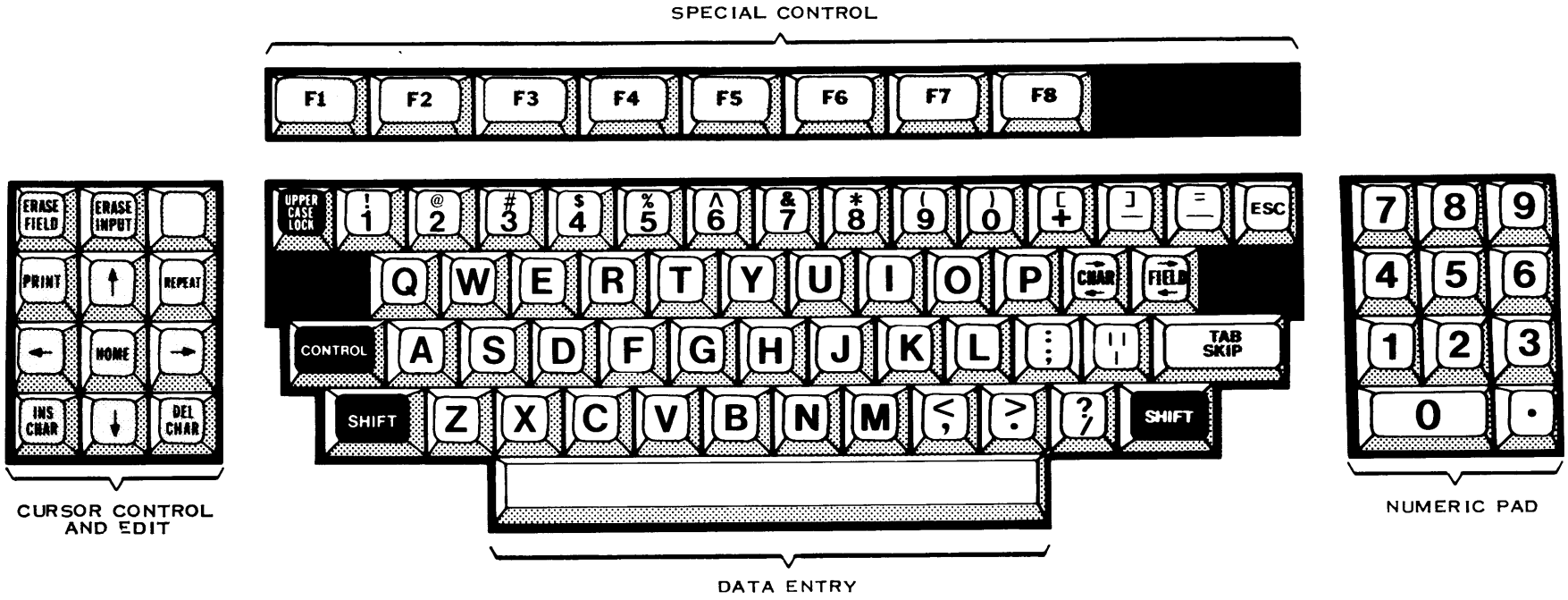
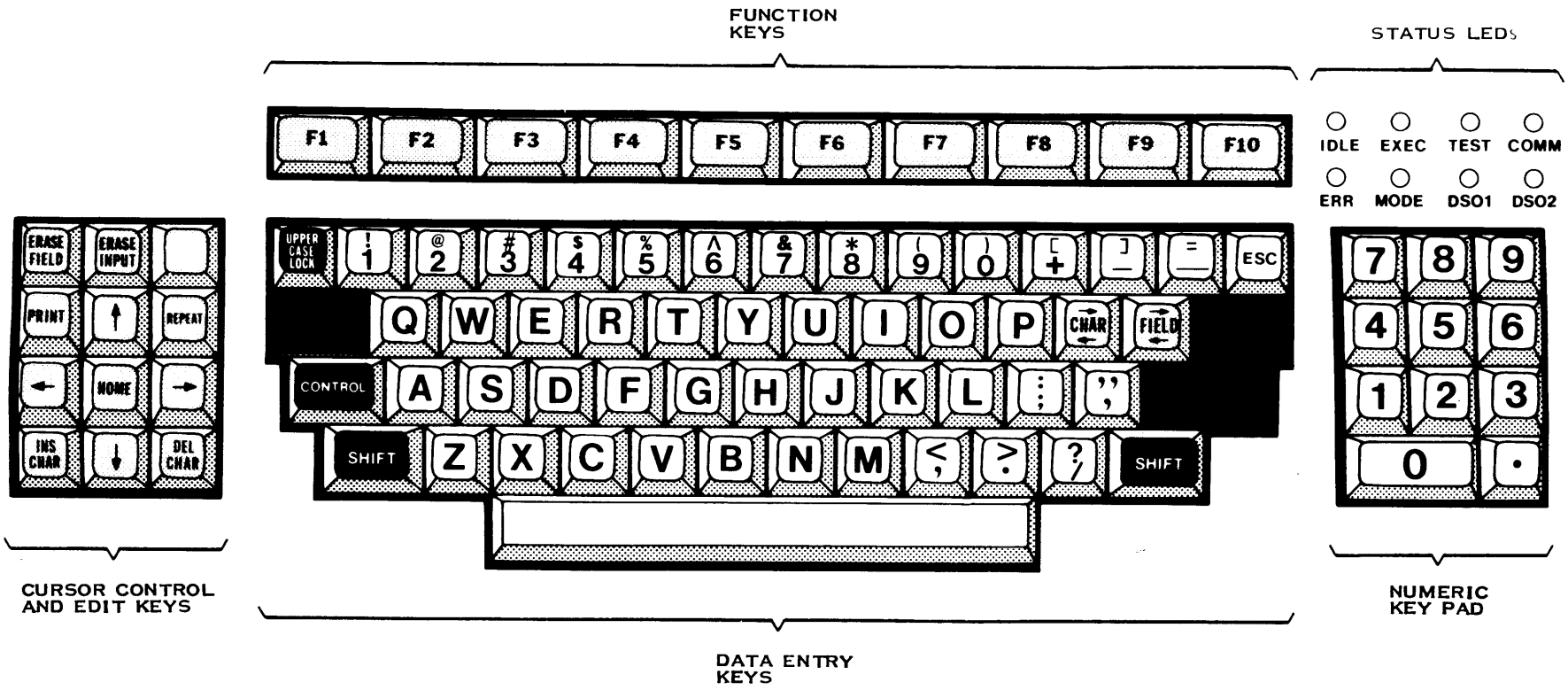
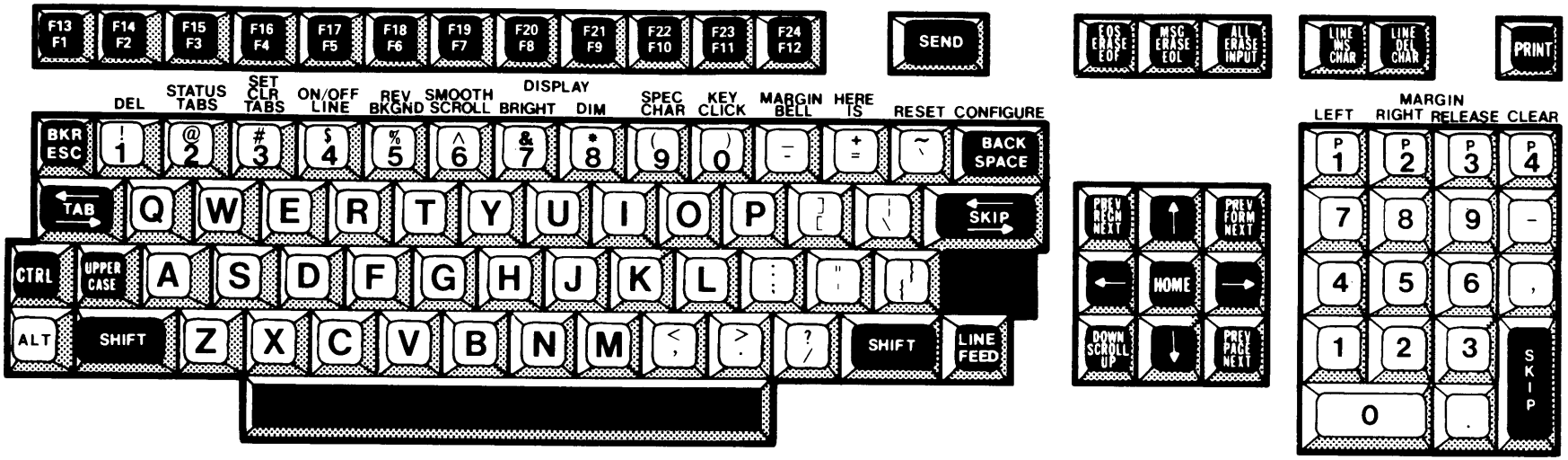


Figure A-1. 911 VDT Standard Keyboard Layout



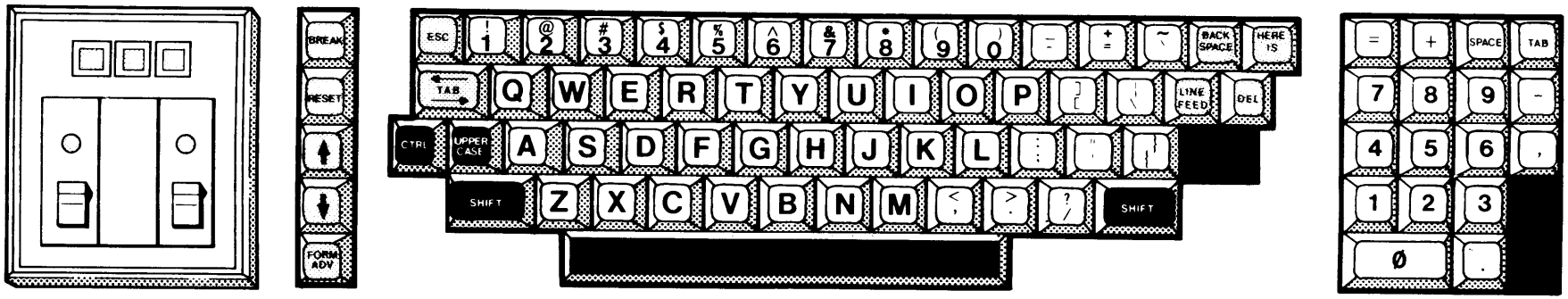
2284734 (10/14)

Figure A-2. 915 VDT Standard Keyboard Layout



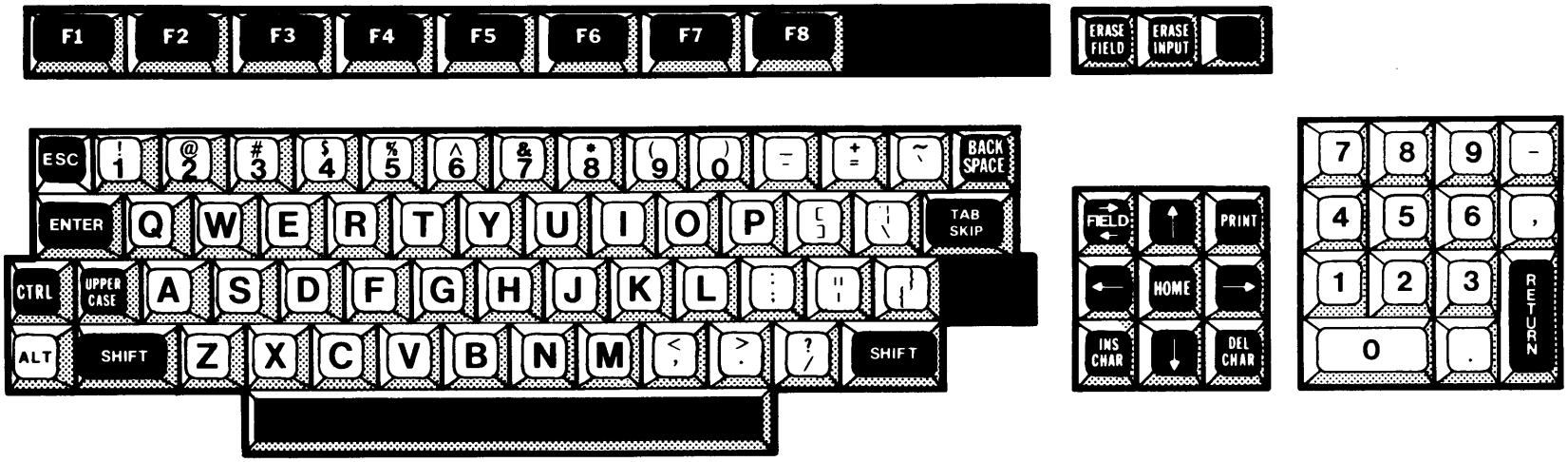
2284734 (11/14)

Figure A-3. 940 EVT Standard Keyboard Layout



2284734 (14/14)

Figure A-6. 820 KSR Standard Keyboard Layout



2284734 (13/14)

Figure A-5. Business System Terminal Standard Keyboard Layout

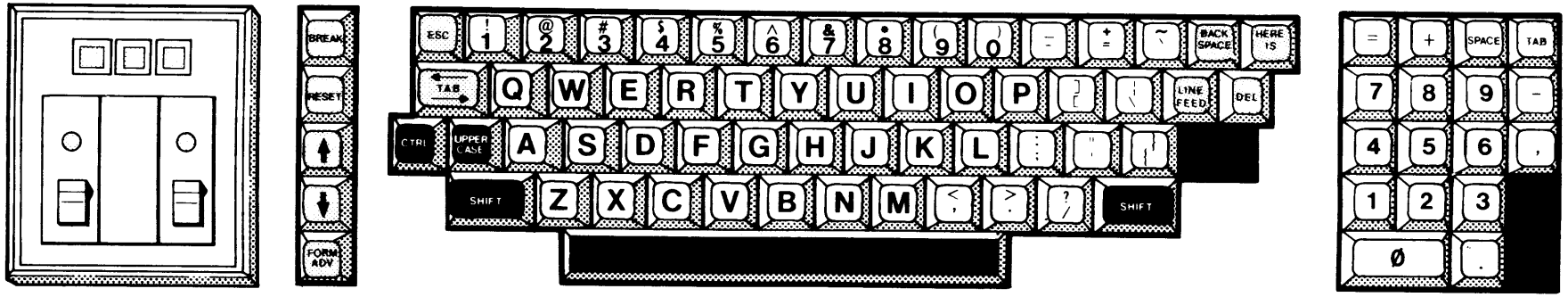


Figure A-6. 820 KSR Standard Keyboard Layout

Appendix B

TIP Compiler Error Messages and Codes

B.1 COMPILER ERROR MESSAGES

This paragraph lists the error messages that the TIP compiler issues. The error number is shown, followed by a letter and the text of the error message. The letter is either W, E, or F: W identifies a warning message, E identifies an error that the compiler may be able to correct, and F identifies a fatal error. The messages are as follows:

1	E	ERROR IN SIMPLE TYPE
2	E	IDENTIFIER EXPECTED
3	E	'PROGRAM' EXPECTED
4	E	')' EXPECTED
5	E	':' EXPECTED
6	E	ILLEGAL SYMBOL
7	E	PARAMETER EXPECTED
8	E	'OF' EXPECTED
9	E	'(' EXPECTED
10	E	ERROR IN TYPE
11	E	'(.' EXPECTED
12	E	'.)' EXPECTED
13	E	'END' EXPECTED
14	E	';' EXPECTED
15	E	INTEGER EXPECTED
16	E	' = ' EXPECTED
17	E	'BEGIN' EXPECTED
18	E	ERROR IN DECLARATION SECTION
19	E	ERROR IN FIELD LIST
20	E	',' EXPECTED
22	E	'. .' EXPECTED
23	E	':' EXPECTED
24	E	';' NOT ALLOWED
25	E	ILLEGAL CHARACTER ENCOUNTERED ON INPUT
40	E	ILLEGAL PARAMETER TYPE — THE TYPE OF A PARAMETER SHOULD BE A PREVIOUSLY DECLARED OR PRE-DEFINED TYPE IDENTIFIER
42	E	STATEMENT TERMINATOR EXPECTED
43	E	STATEMENT EXPECTED
45	W	'EXTERNAL FORTRAN' EXPECTED
46	E	ERROR IN 'CASE' LABEL LIST — ' , ' OR ' : ' EXPECTED
49	E	'ARRAY' EXPECTED
50	E	CONSTANT EXPECTED
51	E	': = ' EXPECTED
52	E	'THEN' EXPECTED

53	E	'UNTIL' EXPECTED
54	E	'DO' EXPECTED
55	E	'TO'/'DOWNTO' EXPECTED
57	E	'FILE' EXPECTED
58	E	ERROR IN FACTOR
60	E	'CHAR' EXPECTED
61	E	UNARY '+'/'–' NOT ALLOWED IN TERM, OR 'NOT' IN BOOLEAN PRIMARY
62	E	EXPRESSION OR ':)' EXPECTED
63	E	':' ',' OR ':)' EXPECTED
64	E	',' OR ':)' EXPECTED
65	E	USE 'DIV' FOR INTEGER '/'
66	E	TYPE IDENTIFIER EXPECTED
67	W	ILLEGAL PARAMETER TYPE—SHOULD BE EITHER A TYPE IDENTIFIER OR A DYNAMIC PARAMETER WITH A '?' UPPER BOUND
68	E	PROGRAM PARAMETERS NOT IMPLEMENTED
69	F	BINARY BOOLEAN OPERATOR EXPECTED
80	W	OPTION IDENTIFIER EXPECTED
81	W	ILLEGAL OPTION IDENTIFIER
82	W	PROGRAM LEVEL OPTION MUST PRECEDE THE PROGRAM HEADING
83	W	ROUTINE LEVEL OPTION MAY NOT BE CONTROLLED HERE
84	W	NULL BODY EXPECTED
85	W	INITCOMMON MAY ONLY BE SPECIFIED AFTER BEGIN
86	W	LOCALS MAY ONLY BE SPECIFIED BEFORE FIRST PROCEDURE DECLARATION
87	E	ROUTINE LEVEL OPTION MUST PRECEDE THE FIRST STATEMENT
101	E	IDENTIFIER DECLARED TWICE
102	F	LOWER BOUND EXCEEDS UPPER BOUND
103	F	IDENTIFIER IS NOT OF APPROPRIATE CLASS
104	E	UNDECLARED IDENTIFIER
105	F	CLASS OF IDENTIFER IS NOT VARIABLE
107	E	INCOMPATIBLE SUBRANGE TYPES
108	E	FILE NOT ALLOWED HERE
109	E	TYPE OF IDENTIFIER MUST BE ARRAY OR SET
110	E	TAGFIELD MUST BE SCALAR OR SUBRANGE
111	E	RECORD VARIANT CONSTANT INCOMPATIBLE WITH TAGFIELD TYPE
117	W	UNSATISFIED FORWARD REFERENCE TO A TYPE IDENTIFIER OF A POINTER
119	E	';' EXPECTED (PARAMETER LIST NOT ALLOWED)
120	E	FUNCTION RESULT MUST BE SCALAR, SUBRANGE, OR POINTER
121	E	FILE VALUE PARAMETER NOT ALLOWED
122	E	';' EXPECTED (FUNCTION RESULT NOT ALLOWED)
123	E	FUNCTION RESULT EXPECTED
126	F	IMPROPER NUMBER OF PARAMETERS
127	F	TYPE OF ACTUAL PARAMETER DOES NOT MATCH FORMAL PARAMETER
128	E	PARAMETER INCOMPATIBLE WITH PREVIOUS PARAMETER (DYNAMIC ARGUMENT)
129	E	TYPE CONFLICT OF OPERANDS IN AN EXPRESSION
130	F	EXPRESSION IS NOT OF SET TYPE
131	W	INTEGER OPERANDS CONVERTED TO REAL FOR '/'
132	W	BASE TYPE OF SET IS LONG INTEGER

133	W	LONG INTEGER ELEMENT IN SET
134	F	ILLEGAL TYPE OF OPERANDS
135	F	TYPE OF EXPRESSION MUST BE BOOLEAN
136	F	SET ELEMENT TYPE MUST BE SOME ENUMERATION TYPE
137	F	SET ELEMENT TYPE NOT COMPATIBLE
138	F	TYPE OF VARIABLE IS NOT ARRAY
139	F	INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION
140	F	TYPE OF VARIABLE IS NOT RECORD
141	F	TYPE OF VARIABLE IS NOT POINTER
142	F	TYPE OF VARIABLE IS NOT FUNCTION
143	F	INCOMPATIBLE "FOR" EXPRESSIONS
145	F	TYPE CONFLICT IN ASSIGNMENT
146	F	ASSIGNMENT OF FILE NOT ALLOWED
147	F	LABEL TYPE INCOMPATIABLE WITH CASE SELECTOR
148	E	SET BOUNDS OUT OF RANGE
149	E	INDEX TYPE MAY NOT BE INTEGER
150	F	ONLY ASSIGNMENT TO LOCAL FUNCTION ALLOWED
151	F	ASSIGNMENT TO FORMAL FUNCTION IS NOT ALLOWED
152	E	NO SUCH FIELD IN THIS RECORD
153	E	CAN'T TAKE SIZE OF PACKED FIELD
154	F	ACTUAL PARAMETER MUST BE A VARIABLE
155	F	CANNOT ASSIGN TO "FOR" CONTROL VARIABLE
156	E	MULTIDEFINED CASE LABEL
161	W	PROCEDURE OR FUNCTION ALREADY DECLARED AT A PREVIOUS LEVEL
162	E	PROCEDURE OR FUNCTION AGAIN DECLARED FORWARD OR EXTERNAL
163	W	PROCEDURE OR FUNCTION ALREADY DECLARED EXTERNAL
164	W	ROUTINE OR COMMON NAME HAS THE SAME FIRST 6 CHARACTERS AS ONE PREVIOUSLY DECLARED; LINK EDITOR CAN'T DISTINGUISH THEM
165	F	MULTIDEFINED LABEL
166	W	USE OF POINTER TO FILE TYPE NOT RECOMMENDED
167	F	UNDECLARED LABEL
168	F	UNDEFINED LABEL
170	W	PROCEDURE PARAMETERS IN FUNCTIONS CAN CAUSE SIDE EFFECTS
171	E	ERROR IN FIXED PRECISION
172	E	ERROR IN DECIMAL PRECISION
173	E	ERROR IN FIXED OR DECIMAL SCALE FACTOR
177	F	EXPRESSION OPERANDS ARE NOT COMPATIBLE
179	E	STATEMENT TO BE ESCAPED MUST BE A STRUCTURED STATEMENT
180	E	ESCAPING BROTHERS PROCEDURES NOT ALLOWED
181	E	CANNOT "GOTO" INTO A "FOR" OR "WITH" STATEMENT
182	F	"FOR" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
183	F	"CASE" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
184	W	IMPLICIT "FOR" IDENTIFIER ALREADY DECLARED
185	E	RECORD VARIANT LABEL MUST BE NONNEGATIVE AND LESS THAN "SETMAX"
186	E	MULTIDEFINED VARIANT LABEL
187	W	USE OF TYPE TRANSFER IN FUNCTION NOT RECOMMENDED
188	W	USE OF "LOCATION" IN FUNCTION NOT RECOMMENDED

189	W	ASSIGNMENT THROUGH LOCAL POINTER IN FUNCTION MAY CAUSE SIDE EFFECTS
190	W	CALLING A PROCEDURE FROM A FUNCTION MAY CAUSE SIDE EFFECTS
191	W	CALLING AN EXTERNAL FUNCTION FROM A FUNCTION MAY CAUSE SIDE EFFECTS
192	W	ASSIGNING TO A GLOBAL VARIABLE, A REFERENCE PARAMETER, OR THROUGH A GLOBAL POINTER IN A FUNCTION, MAY CAUSE SIDE EFFECTS
193	E	ACCESS TO GLOBAL VARIABLE NOT DECLARED
194	W	FILE CONTAINS POINTERS
195	E	TYPE OF COMMON MAY NOT BE A FILE
196	E	DYNAMIC ARRAYS OR SETS NOT ALLOWED IN FILES, RECORDS, OR COMMONS
197	E	ACTUAL REFERENCE PARAMETER CANNOT BE "PACKED VARIABLE" OR "FOR IDENTIFIER"
198	F	ILLEGAL TYPE TRANSFER
199	W	PASS BY REFERENCE OF "PACKED VARIABLES" IS MACHINE DEPENDENT
201	E	FRACTION EXPECTED
202	E	STRING CONSTANT TOO LONG OR CROSSES A RECORD BOUNDARY
203	E	INTEGER CONSTANT TOO LARGE
205	E	BINARY DIGIT EXPECTED
206	E	EXPONENT EXPECTED
207	E	HEXADECIMAL DIGIT EXPECTED
208	E	ILLEGAL LONG INTEGER CONSTANT
209	W	PRECEDING COMMENT HAS NOT BEEN TERMINATED
210	E	FIELD WIDTH MUST BE OF TYPE INTEGER OR LONGINT
211	E	FRACTION LENGTH MUST BE OF TYPE INTEGER OR LONGINT
212	E	HEX FORMAT ALLOWED ONLY FOR TYPE INTEGER OR LONGINT
213	E	BINARY FORMAT ALLOWED ONLY FOR FIXED TYPE
214	E	F-FORMAT ALLOWED FOR REAL, DECIMAL, OR FIXED ONLY
215	E	RANDOM FILE RECORD NUMBER MUST BE OF TYPE INTEGER OR LONGINT
216	E	READ/WRITE PARAMETER NOT COMPATIBLE WITH FILE TYPE
217	W	GLOBAL REFERENCE PARAMETER IN FUNCTION MAY CAUSE SIDE EFFECTS
218	W	READ/WRITE OF A GLOBAL FILE IN A FUNCTION IS CONSIDERED A SIDE EFFECT
219	E	PARAMETER MUST BE OF TYPE FILE
220	E	PARAMETER MUST BE OF TYPE INTEGER OR LONGINT
221	F	INCORRECT NUMBER OF PARAMETERS IN PREDEFINED PROCEDURE CALL
222	F	INCORRECT NUMBER OF PARAMETERS IN PREDEFINED FUNCTION CALL
223	F	PARAMETER MUST BE OF TYPE POINTER
224	E	PARAMETER MUST BE NON-NEGATIVE INTEGER CONSTANT
225	E	PARAMETER MUST BE A CONSTANT
226	E	MISSING CORRESPONDING VARIANT DECLARATION
227	E	PARAMETER MUST BE A CONSTANT OR A VARIABLE
228	W	MANIPULATION OF GLOBAL FILE IN FUNCTION IS CONSIDERED A SIDE EFFECT

229	F	ILLEGAL TYPE OF PARAMETER IN PREDEFINED FUNCTION CALL
230	F	ILLEGAL TYPE OF PARAMETER IN PREDEFINED PROCEDURE CALL
231	E	PARAMETER MUST BE AN INTEGER OR LONGINT CONSTANT
232	F	ACTUAL PARAMETER HAS WRONG STRING LENGTH
233	W	MANIPULATION OF FILE "INPUT"/"OUTPUT" NOT RECOMMENDED
234	F	CANNOT TAKE LOCATION OF PREDEFINED PROCEDURE OR FUNCTION
235	E	DIMENSIONALITY OF ARRAY SMALLER THAN SPECIFIED CONSTANT
236	E	ARRAY COMPONENT TYPES ARE NOT COMPATIBLE
237	E	ARGUMENT MUST BE PACKED ARRAY
238	E	ARGUMENT MUST BE NON-PACKED ARRAY
239	E	FILE ARGUMENT IS THE WRONG FILE KIND
240	W	MAXIMUM REAL PRECISION EXCEEDED
241	E	ARRAY TOO LONG — MORE THAN 32766 BYTES
242	W	RUN-TIME COMPATIBILITY CHECK REQUIRED
243	E	DYNAMIC UPPER BOUND MUST BE GLOBAL VARIABLE OR PARAMETER
244	F	PARAMETER OF WRONG CLASS IN PREDEFINED FUNCTION CALL
245	E	CANNOT TAKE LOCATION OF PACKED FIELD
246	F	RECORD TOO LONG — MORE THAN 32766 BYTES
248	F	FILE OF FILES IS NOT ALLOWED
249	F	"UNIV" PARAMETER CAN HAVE "?" BOUND ON FIRST DIMENSION ONLY
250	F	TOO MANY NESTED SCOPES
251	E	IMPROPER PARAMETER TYPE FOR ACTUAL PROCEDURE PARAMETER
252	E	MISMATCHED REFERENCE PARAMETER FOR ACTUAL PROCEDURE PARAMETER
253	E	VARIABLE PARAMETER EXPECTED FOR ACTUAL PROCEDURE PARAMETER
254	E	IMPROPER NUMBER OF PARAMETERS FOR ACTUAL PROCEDURE
257	F	TOO MANY IDENTIFIERS DECLARED
258	E	GLOBAL VARIABLES NOT ALLOWED WITH LOCALS OPTION
259	F	COLUMN AND STATUS PARAMETERS MUST BE OF TYPE INTEGER
260	W	FORTRAN ARRAY AND RECORD VALUE PARAMETERS WILL BE PASSED BY REFERENCE
261	E	FORTRAN PROCEDURE AND FUNCTION PARAMETERS NOT ALLOWED
262	W	DYNAMIC ACTUAL PARAMETER WILL BE PASSED BY REFERENCE TO FORTRAN
263	W	LENGTH OF TYPE TRANSFER IS GREATER THAN ORIGINAL TYPE'S LENGTH
264	E	POINTER IS TYPE TRANSFERRED TO TYPE OF DIFFERENT SIZE
265	F	PARAMETER OVERFLOW, THE LIMIT IS 127 PARAMETERS AND/OR 508 TOTAL BYTES OF PARAMETER STACK (INCLUDING DYNAMIC SIZE AND UPPER BOUND PARMAMETERS)
266	F	CAN'T PASS PACKED, ODD BYTE ALIGNED ARRAYS (STRINGS) BY VALUE UNLESS THE FORMAL PARAMETER IS A BYTE ELEMENT ARRAY (STRING)
267	F	CAN'T PASS PACKED, ODD BYTE ALIGNED ARRAYS (STRINGS) TO UNIV FORMAL PARAMETER UNLESS THAT UNIV FORMAL IS A BYTE ELEMENT ARRAY (STRING)
268	F	ILLEGAL TYPE TRANSFER OF PACKED, ODD BYTE ALIGNED ARRAY (STRING); MAY ONLY TRANSFER TO FIXED LENGTH BYTE ELEMENT ARRAY (STRING) TYPES

269	F	ILLEGAL TO USE A STRING WHICH REQUIRES LONGINT INDEXING, AS A READ/WRITE PARAMETER (OR AS THE ENCODE/DECODE STRING BUFFER)
270	F	CAN'T PACK OR UNPACK ARRAYS (STRINGS) REQUIRING LONGINT INDEXING
273	E	CAN'T PASS ARRAYS, RECORDS, OR SETS BY VALUE TO C
300	E	DIVISION BY ZERO
301	E	SCALAR "SUCC" OR "PRED" RESULT OUT OF BOUNDS
302	F	INDEX EXPRESSION OUT OF BOUNDS
303	E	VALUE TO BE ASSIGNED IS OUT OF BOUNDS
304	F	SET ELEMENT EXPRESSION IS OUT OF BOUNDS
305	F	LONGINT NOT ALLOWED IN CONSTRUCTED SETS
307	W	COMPARISON TO CONSTANT OUTSIDE RANGE OF EXPRESSION TYPE
308	W	IMPLIED LONGINT TO INTEGER TRUNCATION
309	W	CONSTANT EXPRESSION VALUE OUTSIDE VALID RANGE FOR INTEGER

Appendix C

Run-Time Error Messages and Codes

C.1 RUN-TIME ERROR MESSAGES

For a Pascal task running on DNOS using SCI, the Pascal run-time library routines display error messages on the terminal or in the batch stream listing. These messages are listed below.

Tasks linked with the .TIP.MINOBJ library write the following message to the message file:

CODE = _____ PC = _____ STACK = _____ HEAP = _____

The code value is one of the codes listed below. The PC value is the contents of the program counter when an error occurs, or zero for normal termination. The stack and heap values are hexadecimal numbers of bytes of stack and heap used by the program.

Some error messages specify a statement number, which is the line number within the body of the routine. The routine name is shown in the memory dump; a source listing written with option WIDELIST lists the line numbers in the second column from the left.

The run-time error codes and messages are as follows:

I P000 NORMAL TERMINATION

Explanation:
The execution of the program has terminated without errors at the end of the main program. (This code number is only displayed when the minimal run time is used.)

User action:
(None required)

H PASCAL-P101 MEMORY PARITY ERROR

Explanation:
Memory Parity — The program has been terminated because the operating system was interrupted by a memory parity error interrupt.

User action:
Try the program again. If the problem persists, the computer's memory needs servicing.

US PASCAL-P102 ILLEGAL OPCODE, PC = ?2

Explanation:
The program attempted to execute a machine instruction that is illegal. The PC value shown is the address following the illegal instruction. Possible causes include storing data in the program area, executing a program which had unresolved references in the link-edit, calling a routine in an overlay which has not been loaded, incorrect use of a procedure segment in the link edit, or compiling a program with the 990/12 option and then executing on a smaller processor.

User action:
Determine the cause and correct the program or link-edit control file as required.

US PASCAL-P103 LLEGAL TILINE ADDRESS, WP = ?1, PC = ?2

Explanation:
The operating system has terminated the program with a task error code 3, indicating that an illegal TILINE address has been detected. This is either the address of a non-existent memory or of a non-existent device.

User action:
Check what the program is doing at the indicated PC address and correct it.

U PASCAL-P104 ILLEGAL SUPERVISOR CALL, WP = ?1, PC = ?2

Explanation:
The program executed a supervisor call that referenced an invalid supervisor call code. The first byte of the area referenced by a supervisor call in the supervisor call block is interpreted as the supervisor call code.

User action:
Use the compiler MAP option to verify that the supervisor call block has been layed out in memory as desired. The debugger command ABP can be used to set a breakpoint at the entry to routine SVC, enabling inspection of the call block contents using the LM command. Correct the program to construct a valid call block.

US PASCAL-P105 ADDRESSING ERROR, WP = ?1, PC = ?2

Explanation:
The program has attempted to access a memory location outside the memory region for the task. Two frequent causes for this are accessing an array with an invalid index value and using a pointer with an invalid value.

User action:
The error in the program must be found and corrected. The compiler options CKINDEX and CKPTR may be used to help find the error.

U PASCAL-P106 PRIVILEGED INSTRUCTION, PC = ?2

Explanation:
The program area contains a privileged machine instruction. The PC value shown is the address following the privileged instruction. Pascal programs execute in the nonprivileged mode unless they have been installed as privileged tasks. This error may be the result of storing data in the program area.

User action:
Correct the program.

I PASCAL-P107 TASK KILLED

Explanation:
Execution of the task has been manually terminated with an SCI command (Kill Task or Kill Background Task).

User action:
(none required)

USH PASCAL-P1xx TASK ERROR #?1, WP = ?2, PC = ?3

Explanation:
The operating system has terminated the task execution with the task error code shown as ?1.

User action:
Consult the "DNOS Messages and Codes Reference Manual" for the meaning of the task error code.

U PASCAL-P200 STACK OVERFLOW
P201
P202
P203

Explanation:
The program requires more stack space than has been provided. This normally means that it needs to be given more space. Another cause of this error is a recursive routine that continues to call itself indefinitely.

User Action:
Execute the program again, requesting a larger amount of stack space. If the stack overflow is caused by an unlimited recursion, this will not help and the program must be corrected.

US PASCAL-P300 HALT CALLED

Explanation:
Execution has been terminated by a call to the library routine HALT.

User action:
Consult the operating instructions for the particular program being run.

U PASCAL-P301 HEAP FULL

Explanation:
Heap Full — The program tried to use more heap space than is available. Unless the program has been linked to use a static heap region, get-memory calls are done automatically to obtain more heap space until the full 64K memory region is used up. If this error occurs on execution of the Pascal compiler, it indicates that the routine being compiled is too large for the compiler to be able to handle it.

User action:
For user tasks, either a larger static heap region needs to be linked in, or the program needs to be re-structured to either reduce the task size (for example, by using overlays) or reduce the heap space required (for example, by using PACKED structures, or putting some data in a disk file instead of in memory). If the compiler runs out of heap space, then the routine which it was attempting to compile should be split into two smaller routines.

U PASCAL-P302 DIVIDE BY ZERO

Explanation:
An integer or fixed-point division was attempted with a divisor of zero.

User action:
Correct the program or data.

U PASCAL-P303 WRONG NUMBER OF ARGUMENTS IN FORTRAN CALL

Explanation:
A FORTRAN subroutine or function has been called with the wrong number of arguments.

User action:
Correct the program.

U PASCAL-P304 STACK OVERFLOW IN REENTRANT FORTRAN CALL

Explanation:
Insufficient stack space remained for a call to a reentrant FORTRAN routine in the program.

User action:
Execute the program again, requesting a larger amount of stack space.

U PASCAL-P305 DYNAMIC ARRAY SIZE OUT OF RANGE

Explanation:
Dynamic Array Size Out of Range — The value of the size of a dynamic array was either zero or negative when the routine that contains the array was entered.

User action:
Determine where the invalid size came from and correct the program or data.

U PASCAL-P306 FORTRAN FLOATING POINT ERROR

Explanation:
A floating-point arithmetic error has occurred while executing a FORTRAN routine. The error may be overflow, divide by zero, illegal instruction, or double precision routines required.

User action:
Examine the program to determine the cause and correct the program or data.

US PASCAL-P365 INVALID PACKET POINTER FOR DISPOSE

Explanation:
The DISPOSE routine has been called with a pointer operand that does not contain the address of a valid heap packet.

User action:
Correct the program so that a valid pointer value is used.

US PASCAL-P366 INVALID LENGTH OF PACKET IN DISPOSE

Explanation:
The DISPOSE routine has been called with a pointer operand which points to a heap packet with an invalid packet length word preceding it. This might mean either that the pointer value is incorrect, or that the packet length word has been destroyed, possibly by storing beyond the end of the previous packet in the heap.

User action:
Use the Pascal memory dump to determine the cause of the problem, and correct the program as required.

U PASCAL-P400 CANNOT GET MEMORY

Explanation:
Memory required for the program plus the memory requested for stack and heap exceeds the amount of available memory. The task size (as shown on the link-edit listing) + stack size + heap size + 192 overhead bytes must be less than 65,536 bytes.

User action:
Try to execute the program with smaller stack and/or heap values. If the program gets stack or heap overflow with smaller parameters, then it simply needs more memory than is available, and it will be necessary to either reduce the program size (for example, by using overlays) or reduce the amount of data being held in memory.

U PASCAL-P401 MEMORY RESIDENT TASK NOT LINKED WITH MINOBJ

Explanation:
A Pascal task was installed as memory resident and has been executed a second time without being properly initialized.

User action:
If the task needs to be memory resident, it must be linked with the minimal run-time library:
.TIP.MINOBJ and .TIP.LUNOBJ

U PASCAL-P403 PROCESS EXITED WITH RETURN INSTEAD OF RESUME

Explanation:
A user-created process within a Pascal program has returned from its top-level procedure instead of resuming another process.

User action:
Correct the program.

U PASCAL-P404 "ESCAPE"-FROM-ROUTINE ERROR

Explanation:
An ESCAPE statement that references a routine name has been executed but the named routine cannot be found in the stack. This could indicate that an attempt is being made to escape from a routine which is not currently active, or that the routine was not compiled with the TRACEBACK option on.

User action:
Correct the program so that a valid routine name is used and that the named routine is compiled with TRACEBACK enabled.

U PASCAL-P405 INVALID RELEASE POINTER

Explanation:
The library procedure RELEASE has been called with a pointer operand that does not contain the address of a valid heap packet.

User action:
Correct the program so that a valid pointer value is used.

USH PASCAL-P406 TCA ACCESS ERROR

Explanation:
An error has been returned by one of the SCI interface routines used by the task. This could result from calling procedure STORE$SYN when there is not enough room in the synonym table to hold the new synonym. It could also be a hardware or system I/O error on attempting to access the synonym table.

User action:
Check to see if procedure STORE$SYN is being used improperly. Check the system log file to see if there are any disk errors logged.

U PASCAL-P421 TEXT FILE I/O ERROR: PARAMETER OUT OF RANGE NAME = ?1

Explanation:
A text file utility routine has been called with an invalid argument value. For example, a field width specification which is not greater than zero will produce this error.

User action:
Correct the program.

U PASCAL-P422 FIELD WIDTH TOO LARGE FOR TEXT FILE "?1"

Explanation:
In a WRITE or WRITELN to a text file, a field width is specified which is longer than the logical record length of the file.

User action:
Either modify the program to use a valid field width, or use the CFSEQ command to create a file with a larger logical record length.

U PASCAL-P423 INCOMPLETE DATA IN READ FROM TEXT FILE "?1"

Explanation:
A READ or READLN from a text file obtained a value that is syntactically incomplete. For example, a "+" or "–" which is not followed by a digit is an incomplete number.

User action:
Correct the data in the file and execute the program again.

U PASCAL-P424 INVALID CHARACTER IN FIELD READ FROM TEXT FILE "?1"

Explanation:
A READ operation on a text file obtained a character that is invalid for the specified data type. For example, a decimal point (.) is invalid in an integer value.

User action:
Correct the data in the file and execute the program again.

U PASCAL-P425 VALUE TOO LARGE IN READ FROM TEXT FILE "?1"

Explanation:
A READ operation on a text file obtained a value too large to be represented as the type specified. For example, 33000 is too large to be represented as an INTEGER type variable.

User action:
Either correct the value in the file being read or change the type specification to a type compatible with the file.

U PASCAL-P426 READ PAST END-OF-FILE ON TEXT FILE "?1"

Explanation:
An unformatted READ has been attempted from a text file, and while skipping blanks and end-of-lines, end-of-file was reached before finding any data.

User action:
The program should use function EOF to test for end-of-file, and not try to read beyond it.

U PASCAL-P427 FIELD EXCEEDS RECORD SIZE ON TEXT FILE "?1"

Explanation:
The field specified for a formatted READ from or WRITE to a text file is longer than the logical record length of the file.

User action:
Either correct the program to use a valid field width or use the CFSEQ command to create a file with a larger logical record length.

US PASCAL-P440 ILLEGAL INSTRUCTION (?1) ENCOUNTERED BY FLOATING POINT INTERPRETER AT LOCATION ?2

Explanation:
The floating point interpreter encountered an instruction which it did not recognize. Either CODEGEN wrote bad object code or the program code area has been modified, possibly by storing data at the wrong location. (Table C-1 lists the instruction codes.)

User action:
Determine the source of the invalid instruction and correct as required.

U PASCAL-P441 FLOATING POINT DIVISION BY ZERO AT LOCATION ?2#
(INSTRUCTION = ?1)

Explanation:
The divisor of a division operation on REAL numbers was zero.

User action:
Correct the program or data.

U PASCAL-P442 FLOATING POINT OVERFLOW AT LOCATION ?2 (INSTRUCTION = ?1)

Explanation:
A floating point operation has resulted in a value that is too large to be represented (magnitude greater than 1E+75). (Table C-1 lists the instruction codes.) This message is also issued when an attempt is made to convert a REAL data item greater than 32,767 to INTEGER.

User action:
Correct the program or data.

Table C-1. Floating-Point Instruction Values

Single-Precision Instruction	Double-Precision Instruction	Operation
0C00	0C01	Convert REAL to INTEGER
0C02	0C03	Negate
0C04	0C05	Convert REAL to LONGINT
0C06	0C07	Convert LONGINT to REAL
0C40–0C7F	0E40–0E7F	Add
0C80–0CBF	0E80–0EBF	Convert INTEGER to REAL
0CC0–0CFF	0EC0–0EFF	Subtract
0D00–0D3F	0F00–0F3F	Multiply
0D40–0D7F	0F40–0F7F	Divide
0D80–0D8F	0F80–0FBF	Load
0DC0–0DFF	0FC0–0FFF	Store

US PASCAL-P443 DOUBLE PRECISION FLOATING POINT SUPPORT NEEDS TO BE LINKED IN

Explanation:
The runtime routines for double precision REAL values were not linked with a program that attempts to perform a double precision REAL operation.

User action:
Add the following to the link control file: INCLUDE (FL$DBL)

U PASCAL-P444 FLOATING POINT UNDERFLOW AT LOCATION ?2 (INSTRUCTION = ?1)

Explanation:
A floating point operation has resulted in a value that is too small to be represented (magnitude smaller than 1E – 78). (Table C-1 lists the instruction codes.)

User action:
Correct the program or data.

U PASCAL-P460 ARGUMENT TOO LARGE FOR ?1

Explanation:
The value of the argument of a call to the SIN, COS or EXP function is too large. For the SIN and COS the absolute value of the operand must be less than 2E + 55.

User action:
Correct the program or data.

U PASCAL-P461 NEGATIVE ARGUMENT FOR ?1

Explanation:
The square root or logarithm function has been called with a parameter value less than zero.

User action:
Correct the program or data.

U PASCAL-P464 OVERFLOW IN ?1

Explanation:
A floating-point division by zero has occurred inside one of the mathematical functions in the run-time library.

User action:
Correct the program or data.

S PASCAL-P465 DIVISION BY ZERO IN ?1

Explanation:
Division by 0 has been attempted inside one of the library math functions.

User action:
Submit a Software Trouble Report since this should not be possible.

U PASCAL-P481 ATTEMPT TO READ PAST EOF ON FILE "?1" (ACCESS NAME ?2)

Explanation:
The program has attempted to read from a file that is positioned at the end.

User action:
The program should test for end-of-file by calling function EOF. If it is desired to read a file that follows the EOF of the same medium, call procedure SKIPFILES to position the file at the beginning of the next file.

U PASCAL-P482 ATTEMPT TO SKIP BEYOND EOM ON FILE "?1" (ACCESS NAME ?2)

Explanation:
An attempt has been made to skip beyond the end-of-medium (EOM) in a call to procedure SKIPFILES.

User action:
Correct the program to prevent the call. When EOF is true following execution of SKIPFILES, the file is at EOM and no further call to SKIPFILES should be made.

U PASCAL-P483 FILE "?1" NOT OPENED FOR READING

Explanation:
Not Opened for Reading — A READ operation was attempted on a file that has not been opened for reading.

User action:
Correct the program to execute a RESET operation on the file before reading, or either a RESET or EXTEND operation if the file is a RANDOM file.

U PASCAL-P4A1 P4A2 RUN-TIME COMPATIBILITY CHECK AT LINE ?1, INDEX BOUNDS ARE ?2 AND ?3

Explanation:
Run-Time Compatibility Check: Index — The length of a dynamic array is not compatible with the valid length requirements of the operation being attempted. For example, if a dynamic array is assigned to another array, then its actual index bounds must be the same as the other array. This error could also occur on a string comparison, or passing a dynamic array as a parameter to a routine requiring an array of some constant length or two dynamic array parameters of the same length.

User action:
Correct the program.

U PASCAL-P4A3 RUN-TIME COMPATIBILITY CHECK AT LINE ?1, SET BOUNDS ARE ?2 AND ?3

Explanation:
Run-Time Compatibility Check: Set Bound — The size of a dynamic set is not compatible with the valid length requirements of the operation being attempted. A dynamic set assigned to another set or used in a set expression must have the same actual size as the other operand.

User action:
Correct the program.

U PASCAL-P4B1 P4B2 INDEX OUT OF RANGE AT LINE ?1, VALUE = ?2, RANGE = ?3..?4

Explanation:
Run-Time Error: Index — An array index was set to a value outside the range specified for the array. This message only occurs when the CKINDEX compiler option applied when the code was compiled.

User action:
Correct the program to use a valid index.

U PASCAL-P4B3 SUBRANGE VIOLATION AT LINE ?1, VALUE = ?2, RANGE = ?3..?4

Explanation:
Run-time Error: Subrange — A variable or field of subrange or scalar type has been assigned a value outside of the subrange. This message only occurs when the CKSUB compiler option applied when the code was compiled.

User action:
Correct the program to assign a valid value.

U PASCAL-P4B4 LONGINT SUBRANGE VIOLATION AT LINE ?1, VALUE = ?2, RANGE = ?3..?4

Explanation:
Run-Time Error: LONGINT Subrange — A variable or field of type subrange of LONGINT was set to a value outside of the subrange. This message only occurs when the CKSUB compiler option applied when the code was compiled.

User action:
Correct the program to assign a valid value.

U PASCAL-P4B5 P4B6 SCALAR BOUND VIOLATION AT LINE ?1, VALUE = ?2, RANGE = ?3..?4

Explanation:
Run-Time Error: Scalar Bound — A scalar value outside the defined range has been generated. This message only occurs when the CKSUB compiler option applied when the code was generated. Executing function PRED on the smallest value in the range or function SUCC on the largest value in the range causes this error.

User action:
Correct the program.

U PASCAL-P4B8 CASE ALTERNATIVE ERROR AT LINE ?1, VALUE = ?2

Explanation:
Run-Time Error: Longint Case Alternative — The value of the selector expression of a CASE statement is not equal to any of the case labels; the OTHERWISE clause was omitted; and the type of the selector expression is LONGINT.

User action:
Correct the program.

U PASCAL-P4B9 SET BOUND VIOLATION AT LINE ?1, VALUE = ?2, RANGE = ?3..?4

Explanation:
Run-Time Error: Set Bound — A set element that is not within the range of the base type of the set was specified. This message only occurs when the CKSET compiler option applied when the code was compiled.

User action:
Correct the program.

U PASCAL-P4BA NIL POINTER AT LINE ?1

Explanation:
Run-Time Error: NIL Pointer — The program has attempted to use a pointer variable that has a value of NIL. This message only occurs when the CKPTR compiler option applied when the code was compiled.

User action:
Correct the program.

U PASCAL-P4BB INVALID RECORD TAG FIELD AT LINE ?1

Explanation:
Run-Time Error: Record Variable — A reference to the variant portion of a record is inconsistent with the current value of the tagfield. This message only occurs when the CKTAG compiler option applied when the code was compiled.

User action:
Correct the program to use a valid field reference, or turn off the CKTAG option if the reference to a field in a different varient is intentional.

U PASCAL-P4C0 CASE ALTERNATIVE ERROR AT LINE ?1, VALUE = ?2

Explanation:
"CASE" Alternative Error — The selector expression of a CASE statement has a value that is not equal to any of the CASE labels, and there is no OTHERWISE clause.

User action:
Correct the program.

U PASCAL-P4C3 ARITHMETIC OVERFLOW AT LINE ?1

Explanation:
Overflow — An arithmetic operation of INTEGER, FIXED or DECIMAL type operands has resulted in an overflow. Normally this error occurs only if the CKOVER compiler option applied when the code was compiled.

User action:
Correct the program or data.

U PASCAL-P4C7 PRECISION CHECK FAILURE AT LINE ?1

Explanation:
Precision Check Failure — An arithmetic operation of FIXED or DECIMAL type operands has occurred resulting in the loss of one or more most significant digits. This message appears only when the CKPREC compiler option applied when the code was compiled.

User action:
Correct the program or data.

U PASCAL-P4CA "ASSERT" FAILED AT LINE ?1

Explanation:
An ASSERT statement in the Pascal program had its Boolean expression evaluate to FALSE.

User action:
Either the program or the ASSERT statement is incorrect and should be fixed.

U PASCAL-P4CF FATAL COMPILATION ERRORS IN ROUTINE "?1"

Explanation:
An attempt has been made to execute a routine which had fatal compilation errors.

User action:
Correct the source program and re-compile.

U PASCAL-P501 FILE "?1" OPEN ERROR — LUNO #?2 NOT ASSIGNED

Explanation:
Attempted to open a Pascal file with a RESET call, but the logical unit number for the file has not been assigned.

User action:
Use the AL command to assign the logical unit number.

USH PASCAL-P5xx CAN'T OPEN FILE ?1, ACCESS NAME ?2 — INTERNAL SVC CODE 00xx

Explanation:
The operating system has reported an error while executing the RESET or EXTEND procedure to open a file. Some of the more common SVC code values and their meanings are listed below. For a more detailed description or for codes not listed here, consult the table of I/O SVC errors in the appropriate operating system manual.

0001	LUNO not assigned
0006	Device time-out or abort
0007	Device error
0021	Invalid device or volume name
0027	Pathname does not exist
003B	File already open with exclusive access
0043	Magnetic tape unit offline
0072	Pathname is a directory or program file
009A	Invalid directory pathname

U PASCAL-P5FF FILE "?1" (ACCESS NAME ?2) OPEN ERROR —
P6FF FILE TYPE AND ACCESS METHOD CONFLICT

Explanation:
Upon opening a Pascal file with a call to REWRITE or EXTEND, it is found that the file already exists, and the file type is in conflict with the type of file specified by the Pascal program.

User action:
Either delete the file and allow the program to auto-create it to the correct file type, or modify the program to match the file.

U PASCAL-P600 FILE "?1" (ACCESS NAME ?2) OPEN ERROR — ELEMENT SIZE GREATER THAN RECORD LENGTH

Explanation:
Upon opening a Pascal file with a call to REWRITE or EXTEND, it is found that the file already exists, and the logical record length it was created for is less than the size of the record specified by the Pascal program.

User action:
Either delete the file and allow the program to auto-create it the correct size, or modify the program to match the file.

USH PASCAL-P6xx CAN'T OPEN FILE ?1, ACCESS NAME ?2 — INTERNAL SVC CODE 00xx

Explanation:
The operating system has reported an error while executing the REWRITE or EXTEND procedure to open a file. Some of the more common SVC code values and their meanings are listed below. For a more detailed description or for codes not listed here, consult the table of I/O SVC errors in the appropriate operating system manual.

0001 LUNO not assigned
000F Device already in use
001A Disk volume is write-protected
0021 Invalid device or volume name
0027 Pathname does not exist
003B File already in use
0043 Magnetic tape unit offline
00A1 Directory is full
00E0 Disk volume is full

USH PASCAL-P7xx READ ERROR ON FILE ?1, ACCESS NAME ?2—INTERNAL SVC CODE 00xx

Explanation:
The operating system has reported an error while executing the READ or READLN procedure. Some of the more common SVC code values and their meanings are listed below. For a more detailed description or for codes not listed here, consult the table of I/O SVC errors in the appropriate operating system manual.

0006 Device time-out or abort
0030 Read beyond last record — for a random file, this indicates an invalid record number; for a sequential or text file, there is no EOF at the end of the file.
0043 Magnetic tape unit offline

U PASCAL-P800 FILE "?1" NOT OPENED FOR WRITING

Explanation:
The Pascal WRITE or WRITELN routine has been called without first calling REWRITE or EXTEND to open the file.

User action:
Correct the program to open the file before writing.

USH PASCAL-P8xx WRITE ERROR ON FILE ?1, ACCESS NAME ?2 — INTERNAL SVC CODE 00xx

Explanation:
The operating system has reported an error while executing the WRITE or WRITELN procedure. Some of the more common SVC code values and their meanings are listed below. For a more detailed description or for codes not listed here, consult the table of I/O SVC errors in the appropriate operating system manual.

0006 Device time-out or abort
0010 Device operation manually aborted
0043 Magnetic tape unit offline
0044 Magnetic tape without write ring
00DA Secondary allocation table full — disk volume is too full or too fragmented
00E0 Disk volume is full

USH PASCAL-P9xx CANNOT LOAD OVERLAY — INTERNAL SVC MESSAGE CODE 14xx

Explanation:
The operating system has reported an error while attempting to load an overlay. Some of the more common SVC code values and their meanings are listed below. For a more detailed description or for codes not listed here, consult the table of internal SVC errors in the *DNOS Messages and Codes Reference Manual* for code 14xx.

1411 ID word error (hardware failure)
1413 disk controller timeout (hardware)
14FF see P9FF below

U PASCAL-P9FF CANNOT LOAD OVERLAY — INTERNAL SVC MESSAGE CODE 14FF

Explanation:
The overlay loader, OVLY$, has been called with an overlay number which cannot be found on the program file.

User action:
Use the MPF command to verify which overlays have been defined, and correct either the program or the link control to be consistent.

C.2 PASCAL INTERNAL CODES

If a DNOS system does not have the message file .S$MSG.PASCAL, a TIP error yields the following message:

PASCAL — INTERNAL CODE >xxxx

The internal code is a hexadecimal value that corresponds to an error code, as follows:

Internal Code	Error Code
>000B	P101
>000C	P102
>000D	P103
>000E	P104
>000F	P105
>0010	P106

Internal Code	Error Code
>0011	P107
>001C	P1xx
>001D	P200
>001E	P300
>001F	P301
>0020	P302
>0021	P303
>0022	P304
>0023	P305
>0024	P306
>0026	P365
>0027	P366
>0028	P600
>0029	P800
>002A	P501
>002D	P5xx
>002E	P6xx
>002F	P7xx
>0030	P8xx
>0031	P9xx
>0032	P400
>0033	P401
>0035	P403
>0036	P404
>0037	P405
>0038	P406
>0053	P421
>0054	P422
>0055	P423
>0056	P424
>0057	P425
>0058	P426
>0059	P427
>0072	P440
>0073	P441
>0074	P442
>0075	P443
>0076	P444
>0092	P460
>0093	P461
>0096	P464
>0097	P465
>00B3	P481
>00B4	P482
>00B5	P483
>00D3	P4A1
>00D4	P4A1
>00D5	P4A3

Internal Code	Error Code
>00E3	P4B1
>00E4	P4B1
>00E5	P4B3
>00E6	P4B4
>00E7	P4B5
>00E8	P4B5
>00EA	P4B8
>00EB	P4B9
>00EC	P4BA
>00ED	P4BB
>00F2	P4C0
>00F5	P4C3
>00F9	P4C7
>00FC	P4CA
>0101	P4CF

C.3 RUN-TIME ERROR MESSAGES, LUNO I/O

When the task is linked with the library .TIP.LUNOBJ, an error message is written on the message file and the output file without an error code. The following list shows the error messages in alphabetical order and the corresponding error code. After determining the error code, consult the list in paragraph C.1 for the explanation.

P105	ADDRESSING ERROR
P4CA	"ASSERT" FAILED
P481	ATTEMPT TO READ PAST EOF
P482	ATTEMPT TO SKIP BEYOND EOM
P483	FILE "_____" NOT OPENED FOR READING
P800	FILE "_____" NOT OPENED FOR WRITING
P400	CANNOT GET MEMORY
P9xx	CANNOT LOAD OVERLAY
P4C0	"CASE" ALTERNATIVE ERROR
P302	DIVIDE BY ZERO
P305	DYNAMIC ARRAY SIZE OUT OF RANGE
P404	ESCAPE-FROM-ROUTINE ERROR
P4CF	FATAL COMPILATION ERRORS
P441	FLOATING POINT ERROR—DIVISION BY ZERO
P440	FLOATING POINT ERROR—ILLEGAL INSTRUCTION
P442	FLOATING POINT ERROR—OVERFLOW
P443	FLOATING POINT ERROR—RELINK FOR EXTENDED PRECISION
P444	FLOATING POINT ERROR—UNDERFLOW
P303	FTN ARGUMENT ERROR

P306	FTN FLOATING POINT ERROR
P304	FTN STACK OVERFLOW
P300	HALT CALLED
P301	HEAP FULL
P102	ILLEGAL OPCODE
P104	ILLEGAL SUPERVISOR CALL
P103	ILLEGAL TILINE ADDRESS
P366	INVALID LENGTH OF PACKET
P9FF	INVALID OVERLAY NUMBER
P365	INVALID PACKET POINTER
P405	INVALID RELEASE POINTER
P460	MATH PACK ERROR—ARGUMENT TOO LARGE
P465	MATH PACK ERROR—DIVISION BY ZERO
P461	MATH PACK ERROR—NEGATIVE ARGUMENT
P464	MATH PACK ERROR—OVERFLOW
P463	MATH PACK ERROR—WRONG NUMBER OF ARGUMENTS
P101	MEMORY PARITY
P000	NORMAL TERMINATION
P500	OPEN ERROR—ELEMENT SIZE > RECORD LENGTH (on a RESET)
P600	OPEN ERROR—ELEMENT SIZE > RECORD LENGTH (REWRITE or EXTEND)
P5xx	OPEN ERROR—STATUS = xx (on a RESET or RANDOM EXTEND)
P5FF	OPEN ERROR—FILE TYPE AND ACCESS METHOD CONFLICT (on RESET)
P6xx	OPEN ERROR—STATUS = xx (REWRITE or EXTEND)
P6FF	OPEN ERROR—FILE TYPE AND ACCESS METHOD CONFLICT (REWRITE or EXTEND)
P4C3	OVERFLOW
P4C7	PRECISION CHECK FAILURE
P106	PRIVILEGED INSTRUCTION
P7xx	READ ERROR — STATUS = xx
P4A1	RUN-TIME COMPATIBILITY CHECK: INDEX
P4A2	RUN-TIME COMPATIBILITY CHECK: LONGINT INDEX
P4A3	RUN-TIME COMPATIBILITY CHECK: SET BOUND
P4B1	RUN-TIME ERROR: INDEX
P4B8	RUN-TIME ERROR: LONGINT CASE ALTERNATIVE
P4B2	RUN-TIME ERROR: LONGINT INDEX
P4B6	RUN-TIME ERROR: LONGINT SCALAR BOUND
P4B4	RUN-TIME ERROR: LONGINT SUBRANGE
P4BA	RUN-TIME ERROR: NIL POINTER
P4BB	RUN-TIME ERROR: RECORD VARIABLE
P4B5	RUN-TIME ERROR: SCALAR BOUND
P4B9	RUN-TIME ERROR: SET BOUND
P4B3	RUN-TIME ERROR: SUBRANGE
P200	STACK OVERFLOW

P1xx	TASK ERROR #xx
P107	TASK KILLED
P406	TCA ACCESS ERROR
P427	TEXT FILE I/O ERROR: FIELD EXCEEDS RECORD SIZE
P422	TEXT FILE I/O ERROR: FIELD WIDTH TOO LARGE
P423	TEXT FILE I/O ERROR: INCOMPLETE DATA
P424	TEXT FILE I/O ERROR: INVALID CHARACTER IN FIELD
P421	TEXT FILE I/O ERROR: PARAMETER OUT OF RANGE
P426	TEXT FILE I/O ERROR: READ PAST END OF FILE
P425	TEXT FILE I/O ERROR: VALUE TOO LARGE
P8xx	WRITE ERROR—STATUS = xx

C.4 DEBUG COMMAND ERROR MESSAGES

The Pascal debug commands of SCI write the following error messages:

U DEBUGGER—0031 — TOO MANY PROCESS RECORDS
The Pascal task consists of more than 100 processes. Either the task is not a Pascal task or task memory has been altered.

U DEBUGGER—0032 — INVALID BREAKPOINT LOCATION
The string entered in response to the WHERE prompt is unrecognizable.

U DEBUGGER—0033 — INVALID STACK FRAME
Either the address of the bottom of the stack frame is not less than the address of the top of the stack frame or the address of the top of the stack frame is not equal to the address of the bottom of the next stack frame. This message results if an SPS, LPS or ABP command is issued before the task has been initialized (immediately following the execution of XPT, specifying the debug mode). To recover from this condition, assign a breakpoint and enter an RT command, then enter the SPS or LPS command. Other causes of this error are when a Pascal debug command is entered for a task that is not a Pascal task, or when task memory has been altered.

U DEBUGGER—0034 — INVALID PROCESS RECORD
Either the process record cannot be found or the structure of the process record is incorrect. Either the task is not a Pascal task or task memory has been altered.

U DEBUGGER—0035 — INVALID ROUTINE NAME
The routine name entered in the command cannot be found. Either the routine does not exist or the routine was compiled with the TRACEBACK option off, or the routine is in an overlay that is not currently in memory.

U DEBUGGER—0036 — NONSTANDARD OR FORTRAN FRAME
The stack frame at the top of the stack is for a nonstandard or FORTRAN routine. The Pascal debug commands require the stack frame to be that of a Pascal routine or an assembly language routine with standard interface.

Appendix D

Estimating Run-Time Sizes

The run-time sizes shown in the descriptions of run-time options in Section 7 are minimum sizes. Table D-1 lists additional features and the amount of additional code required.

Table D-1. Run-Time Size Required for Additional Features

Feature	Size (Bytes)
Input/Output	3300[1]
plus:	
For sequential or text files	600[1]
For sequential files	420
For random files	500
For text files	900[1]
plus:	
For CHAR and string	1970[1]
For INTEGER conversions	1950[1]
For hexadecimal integers	940
For LONGINT conversions	1550
For REAL conversions	4870
For FIXED conversions	5350
For DECIMAL conversions	2200
For BOOLEAN conversions	1400
For SKIPFILES	680
For SETMEMBER	380
plus:	
For miscellaneous I/O library procedures	0 to 330
REAL arithmetic, single-precision	2340
plus:	
For double-precision	1370
Mathematical functions, single-precision	1750 to 3630
plus:	
For double-precision also	2440 to 4850

Table D-1. Run-Time Size Required for Additional Features (Continued)

Feature	Size (Bytes)
Mathematical functions, double-precision only	2440 to 4850
LONGINT arithmetic	0 to 460
FIXED arithmetic	0 to 1770
DECIMAL arithmetic	2170 to 2370
Type conversions	0 to 2760
SET operations	0 to 260
Default heap manager (NEW and DISPOSE)	160[1]
Heap manager with space recovery (NEW and DISPOSE)	430[1]
ASSERT check	210[2]
Optional run-time checks	1310[3]
CASE label checks	350[4]
FORTRAN linkage	410[5]
Overlays (DNOS)	160
PACK	500
UNPACK	770
Procedure linkage (per static nesting level)	42

Notes:

[1] Included in the minimum .TIP.OBJ and .TIP.LUNOBJ.

[2] ASSERT checks require only 40 bytes in addition to the minimum for .TIP.MINOBJ.

[3] Optional run-time checks require only 230 bytes in addition to the minimum for .TIP.MINOBJ.

[4] CASE label checks require only 70 bytes in addition to the minimum for .TIP.MINOBJ.

[5] Includes only the additional bytes from the TIP run-time library. The size of the specified FORTRAN subroutine or function and the associated FORTRAN run-time routines must also be added.

Alphabetical Index

Introduction

HOW TO USE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — Reference to Sections of the manual appear as "Sections x" with the symbol x representing any numeric quantity.

- Appendixes — Reference to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

- Paragraphs — Reference to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.

- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

  Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

  Fx-yy

- Other entries in the Index — References to other entries in the index preceded by the word "See" followed by the referenced entry.

USER'S RESPONSE SHEET

Manual Title: DNOS TI Pascal Programmer's Guide (2270517-9701)

Manual Date: January 1984 **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.

CUT ALONG LINE

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____

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