DNOS

SCI and Utilities Design Document

Part No. 2270513-9701 *B
15 November 1983

TEXAS INSTRUMENTS

MANUAL REVISION HISTORY

DNOS SCI and Utilities Design Document (2270513-9701)

Original Issue	1 August 1981
Revision	1 October 1982
Revision	15 November 1983

The total number of pages in this publication is 570.

2270513-9701

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.

All DNOS Users:

DNOS Concepts and Facilities
2270501-9701

DNOS Operations Guide
2270502-9701

DNOS System Command Interpreter (SCI) Reference Manual
2270503-9701

DNOS Text Editor Reference Manual
2270504-9701

DNOS Messages and Codes Reference Manual
2270506-9701

DNOS Reference Handbook
2270505-9701

DNOS Master Index to Operating System Manuals
2270500-9701

High-Level Language Users:

COBOL Reference Manual
2270518-9701

DNOS COBOL Programmer's Guide
2270516-9701

DNOS Performance Package Documentation
2272109-9701

TI Pascal Reference Manual
2270519-9701

DNOS TI Pascal Programmer's Guide
2270517-9701

FORTRAN-78 Reference Manual
2268681-9701

DNOS FORTRAN-78 Programmer's Guide
2268680-9701

MATHSTAT-78 Programmer's Reference Manual
2268687-9701

FORTRAN-78 ISA Extensions Manual
2268696-9701

TI BASIC Reference Manual
2308769-9701

RPG II Programmer's Guide

Assembly Language Users:

990/99000 Assembly Language Reference Manual
2270509-9701

DNOS Assembly Language Programmer's Guide
2270508-9701

DNOS Link Editor Reference Manual
2270522-9701

DNOS Supervisor Call (SVC) Reference Manual
2270507-9701

Security Managers:

DNOS Security Manager's Guide
2308954-9701

Productivity Tools Users:

DNOS Sort/Merge User's Guide
2272060-9701

DNOS TIFORM Reference Manual
2276573-9701

DNOS Query-990 User's Guide
2276554-9701

DNOS Data Base Management System Programmer's Guide
2272058-9701

DNOS Data Base Administrator User's Guide
2272059-9701

Data Dictionary User's Guide
2276582-9701

DNOS TIPE Reference Manual
2308786-9701

DNOS TIPE Exercise Guide
2308787-9701

DNOS COBOL Program Generator User's Guide
2234375-9701

Communications Software Users:

DNOS DNCS/SNA User's Guide
2302663-9701

DNOS DNCS Operations Guide
2302662-9701

DNOS DNCS 914A User's Guide
2302664-9701

DNOS 3270 Interactive Communications Software (ICS) User's Guide
2302670-9701

DNOS 3780/2780 Emulator User's Guide
2270520-9701

DNOS DNCS System Generation Reference Manual
2302648-9701

DNOS DNCS X.25 Remote File Transfer (RFT) User's Guide
2302640-9701

DNOS Remote Terminal Subsystem (RTS) User's Guide
2302676-9701

Systems Programmers:

DNOS System Generation Reference Manual
2270511-9701

DNOS Systems Programmer's Guide
2270510-9701

DNOS Online Diagnostics and System Log Analysis Tasks User's Guide
2270532-9701

ROM Loader User's Guide
2270534-9701

Source Code Users:

DNOS System Design Document
2270512-9701

DNOS SCI and Utilities Design Document
2270513-9701

DNOS Software Manuals Summary

Concepts and Facilities
Presents an overview of DNOS with topics grouped by operating system functions. All new users (o 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.

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

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

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

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

Diagnostics and System Log Analysis Tasks User's Guide
xplains how to execute the online diagnostic tasks and the system log analysis task and how to inte ret the results.

ader User's Guide
plains how to load the operating system using the ROM loader and describes the error conditio

sign Documents
tain design information about the DNOS system, SCI, and the utilities.

urity Manager's Guide
ribes the file access security features available with DNOS.

PREFACE

The purpose of this document is to provide information pertaining to the organization and operation of the System Command Interpreter and selected utility programs shipped with DNOS. This information is sufficient to enable a system programmer who is not familiar with the code to fix problems that may arise and to make additions and improvements. All changes required to internationalize the programs are discussed.

It is assumed that the reader is familiar with terms and concepts discussed in the DNOS Concepts and Facilities Manual, and Section 3 (Coding Conventions)of the DNOS System Design Document.

Changes made to this version of the manual, since the previous release, are marked with revision bars in the outside margins.

This manual is organized as follows:

Section	
1	How to Use the Design Document - Explains how to use this manual.
2	Conventions and Libraries - Explains conventions used in writing the SCI and utilities code. Contains a summary of routines in S$SYSTEM and UTCOMN, libraries used by SCI and utility programs.
3	Error and Status Message Handling - Describes the components of the DNOS message-handling system -- the message files, utilities that build system message files, and routines that construct messages for display to the user.
4	System Command Interpreter - Discusses the System Command Interpreter.
5	Text Editor - Discusses the Text Editor.
6	System Configuration Utility - Discusses the System Configuration Utility.
7	Operator Interface - Discusses the Operator Interface Subsystem.
8	Spooler - Discusses the Spooler subsystem.
9	File Maintenance Utilities - Discusses the file

maintenance utilities of DNOS.

10 User ID Maintenance - Discusses the DNOS utility for maintaining the set of user IDs on the system.

11 Teleprinter Device Utilities - Describes the utilities used to call, answer, and disconnect teleprinter devices and to examine their characteristics.

12 Debugging - Discusses the tools provided by DNOS for debugging user programs.

13 Volume Utilities - Discusses several utilities that handle disk volumes.

14 Data Structure Pictures - Contains computer generated pictures of data structures used by the utilities.

Appendix

A Keycap Cross-Reference - Discusses the generic keycap names that apply to all terminals that are used for keys on keyboards through out this manual.

B Writing DSEG Position-Independent Code - Explains one technique of writing code that can be shared by more than one task in a program file.

C Task Segments, Procedure Segments and Overlays in .S$UTIL - Presents tables of installed tasks, procedure segments and overlays in the utility program file.

For further information related to the use of DNOS, refer to the manuals shown in the frontispiece.

TABLE of CONTENTS

SECTION 3 ERROR AND STATUS MESSAGE-HANDLING

SECTION 10 USER ID AND ACCESS GROUP MAINTENANCE

SECTION 11 TELEPRINTER DEVICE UTILITIES

SECTION 12 DEBUGGING TOOLS

SECTION 13 VOLUME UTILITIES

LIST of TABLES

LIST of FIGURES

SECTION 1

HOW TO USE THE DESIGN DOCUMENT

This manual is a description of the System Command Interpreter (SCI) and the major DNOS utilities. It is divided into sections according to subsystem or function. SCI is described first, followed by a separate section that describes each of the major utilities. (Not every utility program included in DNOS is documented in this manual.) For an overview of major utilities, skim through this document, reading carefully the overview portion of each section. For details on a particular utility or module within a utility, consult the detailed diagrams and discussion that follow the overview.

Section 3 in the DNOS System Design Document details naming conventions for the DNOS modules. When searching for details about a particular module, use the module name to determine which subsystem description is relevant. For details about special-purpose data structures, consult the section on data structure pictures. Operating system data structures are detailed in the DNOS System Design Document.

This manual assumes that you are familiar, at the user interface level, with the subsystems described here. Refer to the System Command Interpreter (SCI) Reference Manual for details of the utilities at the user interface level.

The symbol > preceding a character string indicates that the characters are hexadecimal digits.

The symbol <> is used to mean not equal.

Data structure pictures in this document are built directly from the templates copied into SCI and utilities source code. The structures are shown with hexadecimal byte counts, special comments, flags, and diagrams.

Most of the special terms used in this document can be found in the glossary in the DNOS Concepts and Facilities manual. Other terms are defined in this document as they are needed. Acronyms for system structures and routine names are introduced at various points throughout the manual. If you choose to read a section from the manual without reading all preceding sections, you may encounter an acronym without an explanation of its meaning. Table 1-1 lists most of the acronyms used in the manual. You may want to refer to this list in conjunction with the glossary for a complete description of a term.

Table 1-1 Acronyms Used in this Manual

Acronym	Meaning
ACC	Accounting record contents
ADR	Alias descriptor record
ADU	Allocatable disk unit
BTA	Buffer table area
CDR	Channel descriptor record
CMD	Command key
CRU	Communications register unit
DEL	Descendant error list
DOR	Directory overhead record
DPD	Disk PDT extension data
DSR	Device service routine
EOF	End-of-file
EOL	End-of-line
FCB	File control block
FDB	File directory block
FDP	File descriptor packet
FDR	File descriptor record
FIR	File information record
IOU	I/O utility task
IPC	Interprocess communication
IPL	Initial program load
IRB	I/O request block
JCA	Job communication area
JSB	Job status block
KDR	Key descriptor record
KSB	Keyboard status block
LDT	Logical device table
LPD	Line printer PDT extension
LUNO	Logical unit number
MRB	Master read/master write buffer
MUW	Multi-unit workspace
NCT	Name correspondence table
PC	Program Counter
PDT	Physical device table
SCA	System communication area
SCI	System Command Interpreter
SCU	System Configuration Utility
SDQ	Spooler device queue

Table 1-1 Acronyms Used in this Manual (Continued)

Acronym	Meaning
SDT	Spooler device table
SPM	Spooler message format
SSB	Segment status block
STA	System table area
TCA	Communication area
TLF	Terminal local file
TSB	Task status block
TTY	Teletypewriter
UDR	User descriptor record
VDT	Video display terminal

SECTION 2

CONVENTIONS AND LIBRARIES

2.1 CONVENTIONS

General coding conventions for DNOS code are discussed in Section 3 of the DNOS System Design Document. Unless noted otherwise in the specific discussion of the utility programs, these conventions are followed in all utilities.

Conventions followed in data segment (DSEG) position-independent routines are discussed in Appendix A of this document.

2.2 S$SYSTEM

S$SYSTEM is a collection of routines used extensively by SCI and the utilities. S$SYSTEM is a shared procedure segment in the S$UTIL program file. Each of the routines is DSEG position-independent (see Appendix A). Only the routines that are not documented in the DNOS Systems Programmer's Guide are covered in detail in this document.

2.2.1 Routines Documented in Systems Programmer's Guide.

The following S$ routines are discussed in detail in the DNOS System Programmer's Guide, in the section titled How to Extend SCI. Refer to that document for further details.

NOTE

S$CMSG AND S$SPLR are not included in the procedure segment S$SYSTEM. They are, however, DSEG position-independent code and are documented in the referenced guide.

Routine	Description
S$BIDT	Allows tasks that are normally bid via the .BID or .QBID primitives to be bid from another task
S$CMSG	Creates a message in a specified buffer
S$CLOS	Closes the terminal local file (TLF)
S$GTCA	Makes the communication area (TCA) available for use by the caller
S$IADD	Adds two 32-bit integers in two's complement form
S$IASC	Converts a 32-bit binary integer into an ASCII text string representing that number
S$IDIV	Divides a 32-bit integer by another 32-bit integer
S$IMUL	Multiplies two 32-bit integers
S$INT	Converts an ASCII text string that represents an integer expression into a 32-bit binary value
S$ISUB	Subtracts 32-bit integers
S$MAPS	Searches the name correspondence table and returns the value of the specified synonym
S$NEW	Initializes the task's run-time data base for use by S$ routines
S$OPNS	Opens a specified file in the same way S$OPEN does but has an additional feature: when the Assign LUNO is performed on the file, a specified user ID and passcode are used for security purposes.
S$OPEN	Opens the terminal local file, or a specified file, for write access
S$PARM	Returns a parameter in the TCA
S$SPLR	Submits a print request from the user's task
S$PTCA	Saves synonym values in the TCA
S$RTCA	Releases the TCA
S$SCOM	Compares two strings and sets the equal and arithmetic greater than bits of the status register

Routine	Description
S$SCPY	Copies the specified string into a specified buffer
S$SETS	Defines or redefines a synonym in the name correspondence table
S$SNCT	Searches the name correspondence table for the synonym that is the immediate successor or predecessor of the specified character string
S$SPLT	Separates elements of a list and returns the first element and the remainder of the list separately
S$STAT	Returns the status of the terminal from which the command processor was activated
S$STOP	Terminates a command processor and returns to SCI (does not terminate SCI)
S$TAD	Returns time and date information maintained by DNOS (ASCII format)
S$TERM	Sets the termination synonyms and terminates the calling task
S$WEOL	Terminates the current line and writes it to the TLF
S$WRIT	Writes a specified text string to the TLF

The remaining routines discussed in this section are not documented in the DNOS Systems Programmer's Guide. The calling sequences are documented in the code.

2.2.2 S$FMT.

S$FMT formats the interactive screen with the full name of the command procedure, and the names and any associated values of field prompts. When S$FMT is called, the SCI variable KWBUFW contains the length of the longest field prompt name. This value allows S$FMT to left-justify the prompts and allow the maximum number of columns to the right of the prompt names.

S$FMT checks for video display terminal/teletypewriter (VDT/TTY) mode and writes to the terminal accordingly.

2.2.3 S$GKEY.

This routine is called with an integer value that represents the position of the field prompt whose value is currently expected to be entered by the user. S$GKEY accepts a new value for any field prompt at or prior to this position on the screen. The values are not verified by S$GKEY. Pointers to the values are stored in the SCI table VALTBL.

Note that the Up Arrow keystroke is processed by S$GKEY. The user is allowed to change values of previously defined field prompts. Any keystroke that moves the cursor to a lower line causes S$GKEY to return to the caller.

2.2.4 S$KEY.

S$KEY sets a name/value pair in the NCT. It does not delete approximately matching names. This routine is called only when it is known that no name resides in the NCT that matches or approximately matches the name being stored.

2.2.5 S$MAPK.

This routine interfaces with the Name Manager to obtain the value of any field prompts stored at the current command procedure nesting that approximately match the character string passed to S$MAPK. S$MAPK builds a structure of the following format:

```
<00><runID><depth level><x><FF>
```

where x is calculated as follows: Subtract one from the binary number that is the ASCII representation of the first character of the string passed to S$MAPK. Call this structure ARGUMENT. Since the name correspondence table (NCT) is in alphabetic order, ARGUMENT is the last entry that can precede the first name in NCT that approximately matches the character string passed to S$MAPK (that is, a name consisting of only the first character).

The following loop is executed:

1. A supervisor call (SVC), is issued to the Name Manager requesting the name/value pair immediately following ARGUMENT.

2. If the Name Manager returns a name that does not exactly match the first six characters of ARGUMENT, the partition of names available to the caller has been exceeded without finding a match. Return a null value to the caller.

3. If the name returned satisfies the approximate matching algorithm when paired with the input character string, then return the value to the caller.

4. Set ARGUMENT to the name that did not approximately match and go to step 1.

By appending the run ID and and depth level to the name, a partition within the NCT is created. Only name and value pairs stored at one depth level are available to command procedures and programs.

2.2.6 S$OPN.

S$OPN is the same as S$OPEN. This alternate label for the entry point exists for historical reasons only.

2.2.7 S$OPNX.

This routine forces an open extend of the specified file. Open extend positions the file at the end-of-file (EOF) after it is opened. S$OPNX has the same interface as S$OPEN.

2.2.8 S$PKEY.

S$PKEY writes a message on the command line of the interactive terminal and waits for a reply.

2.2.9 S$PNCT.

S$PNCT purges the NCT. It calls the Name Manager to delete all name/value pairs that start with the specified character string.

2.2.10 S$RIT.

S$RIT issues an SVC to read information from an interactive terminal.

2.2.11 S$SETK.

S$SETK sets a name/value pair and deletes all names in the NCT that approximately match the name being set.

2.2.12 S$WAIT.

S$WAIT is called by RBID tasks that must be suspended. The calling sequence is the same as for S$TERM. Termination synonyms are set and control is returned to SCI. The calling task is not terminated.

The logic of routine S$WAIT is described in the following metacode:

```
Set Termination Synonyms;
$$RBID=Y;
Close the TLF;
Issue Activate Suspended Task SVC (>07) for parent task (SCI);
Issue Unconditional Suspend SVC (>06) for calling task;

*************************************************************
*  SCI (or another task it bids) executes until SCI issues  *
*  an Activate Suspended Task SVC for this task.            *
*************************************************************

Open the TLF;
Get CODE from call block parameters;
IF $$RBID is non-null
  THEN Return the error (CODE) to the user;
END;
```

2.2.13 S$WIT.

S$WIT issues an SVC to write a specified buffer of text to an interactive terminal.

2.3 UTCOMN

UTCOMN is a library of general-purpose routines used by the utility programs. The routines are written in either Pascal or assembly language. Table 2-1 lists the routines and their functions. The interface routines used by the DNOS error handling system are documented in greater detail in the following paragraphs.

Table 2-1 Functions of UTCOMN Routines

Module/ Routine	Description
UTACNM	Builds the name of the channel or file to which the input file description packet (FDP) is assigned. Callable by Pascal code.
UTCHEK	Checks for errors returned from R$ routines. If there is an error, reports it through UTPUER. Otherwise, returns to the caller.
UTCMPS	Compares two character strings.
UTCVDT	Converts time and date block to one of the following formats: HR:MIN:SEC: WEEKDAY, MONTH DAY, YEAR HR:MIN:SEC:
UTEACT	Collects end-action data and reports it through S$TERM
UTEXIT	A command exit path used by MRFSRF (the processor for Map and Show Relative to File) and LLR (the processor for List Logical Record)
UTGJOB	(Also UTGTSB, UTTINT, UTJINT, UTTHIS) Transverses a job status block (JSB) list or task status block (TSB) list in the running system. Callable only by programs that are hardware-privileged and system tasks
UTLLWT	Calls S$WRIT and S$WEOL to write the specified buffer to the listing file
UTLMSG	Issues an SVC to put a message to the system log file
UTLOGN	Resolves the input pathname, which may be a logical name
UTLWRT	Writes the message buffer (assumed to be prepared by S$CMSG) to the listing file
UTMTBL	Moves tables for the directory sort package
UTPOP	Restores as many as nine registers on exit from a module.

Table 2-1 Functions of UTCOMN Routines (Continued)

Module/ Routine	Description
UTPSER	Pascal interface to call UTSERR
UTPTCH	Patch space
UTPUER	Pascal interface to call UTUERR
UTPUSH	Saves as many as nine registers on entry to a module
UTR$ST	Module containing entry points for the following four S$ routines that are used by Pascal tasks that run in the system job: * S$STOP - Allows tailored cleanup processing. If a routine named CLNUP exists, it is called via BLWP before the SVC to terminate the task is issued. * S$GTCA - Dummy entry point equated to S$STOP * S$PARM - Dummy entry point equated to S$STOP * S$INT - Dummy entry point equated to S$STOP
UTSERR	Reports SVC errors through S$TERM
UTSORT	Module containing callable routines to sort directory entries. The entry point is SORT.
UTUERR	Reports non-SVC errors through S$TERM
UTVERS	Carries version information for use by IPL

2.3.1 UTUERR and UTSERR.

The common routines UTUERR and UTSERR are used by system utilities to do commonly needed processing of utility errors and SVC errors, respectively.

UTSERR is called when an SVC error occurs and the utility is to exit through S$TERM. UTSERR can be used to set up registers for and make the call to S$TERM. The interface to UTSERR is as follows:

```
BLWP  @UTSERR
BYTE  Ra,Rb
```

where:

Ra is a register containing the condition code.
Rb is a register containing a call block pointer.

NOTE

If the SVC is an I/O SVC, the logical unit number (LUNO) must not be released before the call to UTSERR completes.

UTUERR is called when an error message is needed from the UTILITY file of the S$MSG directory. UTUERR makes the appropriate call to S$TERM. The interface to UTUERR is as follows:

```
BLWP  @UTUERR
BYTE  Ra,Rb
```

where:

R0 contains the internal message number.
Ra is a register containing the condition code.
Rb is a register containing a pointer to variable text (0 if none).

If the internal message number supplied to UTUERR is less than >8000, it is treated as an SVC error. Some of the S$ routines return SVC error codes directly without translation. In this case, UTUERR changes the file indicator and passes a call block pointer.

To terminate normally with $$CC set to zero, a system utility clears R0, and executes a BLWP to UTUERR using BYTE R0,R0.

2.3.2 UTPUER.

Routines written in Pascal may call UTPUER to interface to UTUERR. The declaration for UTPUER is as follows:

```
UTPUER(P1:INTEGER;P2:INTEGER;P3:STRING);
```

where:

P1 is the internal message number.
P2 is the condition code.
P3 is a pointer to a variable text string (0 if none).

The string passed to UTPUER is a Pascal string with two byte counts. The first byte count is the maximum size of the string and the second is the actual size of the string.

2.3.3 UTEACT.

A common end-action routine, UTEACT is provided for system utilities. For utilities that require no cleanup of their own, UTEACT can be specified as the end-action routine address. If cleanup is needed, the utility can do so in its own end-action routine and then branch to the common routine via the following instruction:

```
B @UTEACT
```

2.4 USE OF .RBID

The .RBID primitive is used to synchronize alternating execution between SCI and a foreground task. A number of utilities make use of .RBID to alternate execution with SCI. These include the operator interface, the system configuration utility, the text editor, and XANAL. The syntax of .RBID is as follows:

```
.RBID TASK=INT/NAME,[PARMS=(STRING...STRING)],[CODE=INT]
```

The TASK parameter is the installed ID or name of a task on the utility program file (S$UTIL) which is to be bid through .RBID. The optional PARMS parameter is a character string list which is passed to the task each time it is activated. The CODE parameter, which is optional, is an integer value between zero through 255 that the task being bid may access via S$STAT.

For each task that is bid through .RBID, SCI makes an entry into the RBID active table. This table keeps a correspondence of installed IDs to runtime IDs. If the ID of the task bid by .RBID is not in this table, SCI assumes it is an initial bid of the task. If the ID is found in the table, a resume is done for the corresponding runtime ID.

The following is an example of a command procedure that uses the .RBID primitive:

```
.PROC     EX(EXAMPLE PROC)=0,
          INPUT PATHNAME=ACNM("@$EX$IP"),
          OUTPUT PATHNAME(S)=(ACNM),
          PRINT THE FILE?=ELEMENT(Y,N)(NO)
.SYN      $EX$IP="&INPUT PATHNAME"
.SYN      $EX$OP="&OUTPUT PATHNAME"
.SYN      $EX$P="&PRINT"
          .RBID     TASK=>43,PARMS=("@$EX$IP","@$EX$OP","@$EX$P")
.EOP
```

See the documentation on S$WAIT, the routine called by an RBID task to return to SCI.

2.5 NAMING STANDARDS

Names for utility commands, for their prompts, and for synonyms must be chosen carefully so that they are meaningful to the user and consistent with other names already in use. The following paragraphs provide guidelines for naming.

2.5.1 Command Naming Standards.

Commands are named by concatenating the first letter of a verb with the first letter of an object. The verb and object are chosen to describe the command, give a unique name, and blend in with the style of the other commands in the system to maintain a consistent user interface. The verb may be compounded of more than one word, and the object may be compounded of nouns and adjectives. A list of verbs in use is found in Table 2-2, and a list of nouns and adjectives in use is found in Table 2-3. There are a very few exceptions to the first letter rule. One is X for Execute, and another is RW for Rewind. Variants of commands are indicated by a tag letter or part of word. Create File, for example, has six variants: CFSEQ, CFREL, CFKEY, CFIMG, CFPRO, and CFDIR. Two variants of Execute COBOL Compiler exist: XCC and XCCF, the tagged variant meaning to execute in foreground.

Commands must not be named using words with the same first letter that have conflicting meanings. For example, the verb Cancel (C) cannot be used because of conflicts with the current usage of C (Create, Copy, COBOL, etc.). The goal is to avoid human-oriented conflicts in meanings. For example, Create and Copy do not conflict in meanings, as do Create and Cancel. Two words with the same meaning should never be used. In the case of two words with the same first letter, uniqueness should be obtained with compound objects rather than compound verbs wherever possible, such as CIC (Create IPC Channel). Copy/Concatenate, which has a compound verb and no object, was one of the commands for which there was no clean alternative to using a compound verb. It also has no object.

Table 2-2 List of Verbs Used in DNOS Command Names

VERB	LETTER
ACTIVATE	A
ADD	A
ANALYZE	A
ANSWER	A
APPEND	A
ASSEMBLE	A
ASSIGN	A
BACKSPACE	B
BACKUP	B
BEGIN	B
BUILD	B
CALL	C
CHECK	CK
CLEAR	C
CONCATENATE	C
COPY	C
COUNT	C
CREATE	C
DELETE	D
DISPLAY	D
END	E
FIND	F
FORWARD SPACE	F
HALT	H
INITIALIZE	I
INSERT	I
INSTALL	I
KILL	K
LIST	L
MAP	M
MODIFY	M
MOVE	M
PATCH	P
PRINT	P
PROCEED	P
QUIT	Q
READ	R
RECEIVE	R
RECOVER	R
RELEASE	R
REPLACE	R
RESET	R
RESPOND	R
RESTORE	R

Table 2-2 List of Verbs Used in DNOS Command Names (continued)

VERB	LETTER
RESUME	R
REWIND	RW
SAVE	SV
SET	S
SCAN	S
SHOW	S
SIMULATE	S
SNAPSHOT	S
START	S
SUSPEND	S
TEST	T
TRANSFER	T
UNLOAD	U
UNLOCK	U
VERIFY	V
WAIT	WAIT
WRITE	W
EXECUTE	X

Table 2-3 List of Nouns and Adjectives Used as Objects

NOUNS and ADJECTIVES	LETTER
ABSOLUTE	A
ACCESS	A
ALIAS	A
ALL LOGICAL UNITS	AL
ALLOCATABLE DISK UNIT	ADU
ANALYZER	AN
ASSEMBLER	A
ATTRIBUTE	A
CRASH ANALYSIS UTILITY	ANAL
BACKGROUND	B
BACKUP	B
BATCH	B/BATCH
BREAKPOINT(S)	B
BYTE	B
CHANNEL	C
COMMAND DEFINITION TABLE	CDT
COMPILER	C
COMPLETE	C
CONFIGURATION	C
CONFIGURATION UTILITY	CU
CONSISTENCY	C
CONTENTS OF SPECIFIED CRU REGISTER	CRU
CONVERSION	C
COPY	C
COUNT	C
COUNTRY CODE	CC
DATE and TIME	DT
DEBUG/DEBUGGER	D
DEFINITIONS	D
DEVICE	D
DEVICE CONFIGURATION	DC
DIRECTORY	D/DIR
DISK/DISKETTE	D
EDIT/EDITOR/TEXT EDITOR	E
END	E
END OF FILE	EOF
ENTRY	E
ERROR	E
EXECUTION	E
EXPANDED	E
FILE (S)	F
FOREGROUND	F

Table 2-3 List of Nouns and Adjectives Used as Objects (continued)

NOUNS and ADJECTIVES	LETTER
GENERATED/GENERATION	G
GLOBAL LUNO	GL
GROUP	
HORIZONTAL	H
I/O	I
IBM	IBM
IMAGE	I/IMG
INFORMATION	I
INTERFACE	I
INTERNAL	I
IPC	I
JOB (S)	J
KEY INDEXED FILE/KIF	K/KF
LINE(S)	L
LINK EDITOR	LE
LOG/LOGGING	L
LOGICAL	L
LUNO	L
MACRO	M
MARGIN	M
MEDIA	M
MEMBERS	M
MEMORY	M
MESSAGE (S)	M/MSG
MODE	M
MONITOR	M
MULTIFILE	M
NAME (S)	N
NEW	N
OPERATOR	O
OUTPUT	O
OVERLAY	O
PANEL	P
PASCAL	P
PASSCODE	PC
PATCH	P
PATHNAME	P
PERFORMANCE DISPLAY	PD
PRIORITY	P
PROCEDURE	P
PROCESSOR	P
PROGRAM	P/PRO
PROTECTION	P

Table 2-3 List of Nouns and Adjectives Used as Objects (continued)

NOUNS and ADJECTIVES	LETTER
RANDOMLY	R
REALTIME/REAL-TIME	R
RECORD	R
REGISTER (S)	R
RELATIVE	R
RELATIVE RECORD	REL
REQUEST	R
REVERSE	R
RIGHT MARGIN	RM
RIGHTS	R
ROLL	R
SCALING	S
SCHEDULER/SWAP PARAMETERS	SP
SCI	S
SECURITY	S
SEGMENT	S
SEQUENTIAL	S/SEQ
SESSION	S
SET	S
SIMULATED	S
SOFTWARE CONFIGURATION	SC
SPOOLER	S
STACK	S
STATE	S
STATUS	S
STRING	S
SURFACE	S
SYNONYM (S)	S
SYSTEM	S
SYSTEM TABLE SIZES	ST
TABS	T
TASK	T
TERMINAL	T
TRACK	T
UTILITY	U/UTL
USER ID (S)	UI
VALUE	V
VOLUME	V
WORD	W
WORKSPACE	W

2.5.2 Prompt Naming Standards.

The standards for field prompts for commands are that the total number of prompts be minimized and that they be descriptive. For example, if a command requires the runtime ID of a task, the prompt RUN ID should be chosen, because that prompt is in use throughout the command set, and has the required meaning. New prompts should be carefully chosen and universally used.

The use of the phrases ACCESS NAME, PATHNAME, and FILE NAME are also rigorously defined. The definitions are the following:

* PATHNAME or FILE NAME means specifically a disk file. The use of PATHNAME is maintained for compatibility with older software. FILE NAME is the preferred usage.

* ACCESS NAME means a disk file or any other I/O resource.

The phrases LISTING ... and OUTPUT ... are chosen with particular connotation in mind. A LISTING is a report to be viewed by a person, and not be used by another program. Thus, Map Disk produces a LISTING, while Copy/Concatenate produces OUTPUT. The primary intent of the output of the utility should be considered in determining if it is a LISTING or OUTPUT. For example, the fact that programs have been written to process the listing file of the List Directory command should not be cause to call its list OUTPUT.

2.5.3 Naming Synonyms and Logical Names.

Synonyms and logical names used by DNOS utilities should be readily recognized as such. In general, these names are formed using the format $X$Y, where X is the command name and Y is a meaningful abbreviation for the synonym or logical name. The initial $ identifies the synonym as one in use by a DNOS utility. For example, $MDC$DEV might be a synonym for a device name prompt used by the Modify Device Configuration (MDC) command.

2.6 Internationalizing The DNOS Utilities

DNOS is designed to meet the international requirements of the United States as well as most Western European countries and Japan. There are certain SCI commands which may be modified, if so desired, to better fill the needs of the users of a particular country. For instance, the prompts that are displayed when a SCI command begins execution may be modified to be displayed in the user's own language.

To assist internationalization, the DNOS utilities are written to make use of the file based messages as much as possible. In cases where the files are not practical, messages are maintained in a single module for the particular utility. These modules need to be changed as do all the message files. One of the routines used to access the messages files has a message that also needs to be translated; this module is .S$.SOURCE.S$CMSG.

For those commands that have a module with message text, modifications need to be done to the command processor. Table 2-4 lists the SCI commands that may be modified for this purpose. In addition, the IFSVC processor and the SCI module that executes .SVC also need to have message modules changed.

Table 2-4 Command Processors to Change for Internationalizing

Command	Full Command Name
BD	Backup Directory
BDD	Backup Directory to Device
CD	Copy Directory
CKD	Check Disk for Consistency
CKR	Copy Key Indexed File to Sequential File Randomly
CRV	Check and Reset Volume
CSM	Copy Sequential Media
CV	Copy Volume
CVD	Copy and Verify Disk
DCOPY	Disk Copy
DD	Delete Directory
HT	Halt Task
IBMUTL	IBM Diskette Conversion Utility
IDS	Initialize Disk Surface
IO	Install Overlay
IP	Install Procedure
IPS	Install Procedure Segment
IRT	Install Real-Time Task
IT	Install Task
LAG	List Access Groups
LAGFR	List Access Groups File Rights
LB	List Breakpoints
LBP	List Breakpoints-PASCAL
LD	List Directory
LDC	List Device Configuration
LHPC	List Hardcopy Terminal Port Characteristics
LJ	List Jobs
LLN	List Logical Names
LLR	List Logical Record
LM	List Memory
LOM	List Operator Messages
LPS	List Pascal Stack
LSAR	List Security Access Rights
LSB	List Simulated Breakpoints
LSM	List System Memory
MAD	Modify Absolute Disk
MADU	Modify Allocable Disk Unit
MD	Map Disk
MDC	Modify Device Configuration
MKF	Map Key Indexed File
MM	Modify Memory
MOE	Modify Overlay Entry
MPE	Modify Procedure Entry
MPF	Map Program File

Table 2-4 Command Processors to Change for Internationalizing (continued)

Command	Full Command Name
MPI	Modify Program Image
MRF	Modify Relative to File
MS	Modify Synonym
MSE	Modify Segment Entry
MSM	Modify System Memory
MTE	Modify Task Entry
MVI	Modify Volume Information
RCRU	Read Contents of Specified CRU Address
RD	Restore Directory
SAD	Show Absolute Disk
SADU	Show Allocable Disk Unit
SCS	Show Channel Status
SD	Scan Disk
SDT	Show Date and Time
SIR	Show Internal Registers
SJS	Show Job Status
SMM	Show Memory Map
SMS	Show memory Status
SP	Show Panel
SPI	Show Program Image
SPS	Show Pascal Stack
SRF	Show Relative to File
STS	Show Task Status
SVS	Show Volume Status
SWR	Show Workspace Registers
VB	Verify Backup of Directory
VC	Verify Copy of Directory
XJM	Execute Job Monitor
XPD	Execute Performance Display
XSCU	Execute System Configuration Utility
XSGU	Execute System Generation Utility

SECTION 3

ERROR AND STATUS MESSAGE-HANDLING

3.1 OVERVIEW OF THE DNOS MESSAGE-HANDLING SYSTEM

The message-handling system of DNOS involves several sets of message files, utilities to build message files, and routines to construct messages for display to the user. Users who are migrating from DX10 to DNOS should modify utilities to use the DNOS message system for consistency of user interface.

Since the user's interface with DNOS is SCI, the burden of displaying messages to the user lies with SCI. Each of the utilities and support functions generates messages using the same SCI interface, and SCI displays messages to the user. All DNOS utilities that encounter the same condition produce the same message.

Source code is independent of the file structure or message IDs. This allows changing of the message file (deleting, adding, or rearranging message without source code changes.

The DNOS message facility is designed for consistent handling of error and completion messages from all sources, and for ease of internationalization. To attain these design goals, all DNOS language processors and utilities use error messages and termination messages from standard files, call the routine S$TERM to report errors and terminate processing, and isolate the text of all internal messages to a single module.

3.2 SCI INTERFACE FOR MESSAGE HANDLING

The following SCI interface routines and common utility modules are defined for the DNOS message-handling facility:

* S$TERM is used to pass along a message upon task termination.

* S$CMSG is used during processing to format a message to be output from a utility.

* The common modules UTCOMN.SOURCE.UTUERR and UTCOMN.SOURCE.UTSERR are linked with various utilities that use messages in the UTILITY or SVC files.

* UTCOMN.SOURCE.UTPSER is the Pascal-callable interface routine for UTSERR.

* UTCOMN.SOURCE.UTPUER is the Pascal-callable interface routine for UTUERR.

* UTCOMN.SOURCE.UTEACT is used by utility tasks that use common end-action processing.

S$TERM and S$CMSG are documented in the DNOS Systems Programmer's Guide. The UTCOMN routines are documented in the Conventions and Libraries section of this manual.

3.3 THE USE OF $$CC

The synonym $$CC is used to report a completion code. Since the severity of an error depends on the environment in which it occurs, the utility writer decides the degree of severity to report for each error condition that arises. The appropriate value is placed into a register before calling UTSERR or UTUERR. Usually, values of >0000, >4000, or >8000 are set. Some utilities, such as the Link Editor, count warnings or errors and provide that count in the last three digits. Only UTEACT sets $$CC to >C000 under normal circumstances.

The conventional meanings used for the $$CC codes are as follows:

* C000 - fatal - An error which causes the utility to terminate processing of a request without successfully completing the request

* 8000 - nonfatal - An error which causes the utility to omit some part of its usual processing or in some other way complete the user's request without doing the entire operation which the user expects.

* 4000 - warning - A condition has arisen that may or may not cause results to be complete. The user needs to check the output of the utility.

* 0000 - Successful completion.

3.4 MESSAGE CONTENTS

The displayed message has the following format:

SOURCE CATEGORY-ID MESSAGE (ADDITIONAL TEXT)

where:

SOURCE	is a one- to three-character error source indicator.
CATEGORY	is the one- to eight-character name of the DNOS subsystem that generates the error.
ID	is an alphanumeric string.
MESSAGE	is the text of the message.
ADDITIONAL TEXT	is additional text and may be blank. This field is used by SCI to report errors in command procedures. The format of the additional text is as follows: Command procedure name; line number

The combined parts of a message are displayed to the user by the show expanded message (SEM) utility or by SCI when an error has occurred or when some informative message must be displayed. If the command being processed was executed interactively, the message appears on the screen. If the command was executed from a batch stream, the message appears in the batch listing file. If more than one line is required for the message, text is continued on the next line or lines, beginning in the same column as the text of the first line. Margins are set by S$CMSG.

3.4.1 Source Indicator.

The source indicator is one of the character combinations shown in Table 3-1.

Table 3-1 Error Source Indicators

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

3.4.2 Category-ID.

The category of a message is a string of from one to eight characters, such as SVC or LINKER. The category identifies the DNOS subsystem that generated the message.

The message identifier (ID) provides an index into the DNOS Messages and Codes Reference Manual. Internally, this identifier is a key for the key indexed file of expanded messages used by the Show Expanded Message (SEM) command procedure that displays additional information about a particular message.

3.4.3 Message.

The text of the message consists of fixed explanatory information and optional variable text.

3.4.3.1 Fixed Information.

Fixed information resides in a message file. The length of this file-resident portion of the message may be as many as 238 characters, and may include any character except the question mark. The question mark is used as a position marker which is replaced by variable text when the message is processed. Each question mark is followed by a decimal digit from 1 to 9 or an upper case C. The digits 1 to 9 show which variable text element replaces the question mark. (This allows translations of messages to rearrange the variable text within the fixed text of the message.) The upper case C indicates that the remainder of the current line of the message is to be blank-filled, i.e. the C is an effective carriage return, line feed sequence.

3.4.3.2 Variable Text.

Variable text is that part of the displayed message that is not the same each time the message is output. It is determined by the utility that generates the message, and is supplied to SCI along with an identifier of the message file. The variable text is passed to S$CMSG by the calling task in a buffer, with the length in the first byte of the buffer.

The length of an element of variable text may be null, or it may be as many as 235 characters, including variable text delimiters. In a 255-byte buffer for Name Management requests, 20 bytes are reserved for system data.

The semicolon is not a valid variable text character. It is used as a delimiter between elements of variable text. Two consecutive semicolons represent a null variable text element. Variable text may include a pathname, a LUNO, an opcode, or other run-time information.

When a question mark is found in the fixed text, variable text is inserted into the buffer containing a copy of the fixed text. If the specified variable text element is null, the question mark is output without the associated digit. A variable text element that is not referenced in the fixed text is not displayed.

3.4.4 Additional Text.

When additional text appears in the message, the character string passed to S$CMSG by the utility generates the message. In the sense that it is determined at run time, it is variable text, but there is no processing of additional text. The character string is appended to the message constructed from fixed and variable text.

The maximum number of characters of additional text is 255.

3.4.5 Translation Of I/O Errors Encountered By SVCs.

Special handling is performed for the message generated when an SVC passes back an I/O error. When this case is encountered, the message text corresponding to the I/O error is also displayed. For example, the message generated when I/O error 0001 is encountered during an Install Task SVC appears as:

```
USH SVC-0010 THE FOLLOWING I/O ERROR (INTERNAL CODE 0001) WAS
        ENCOUNTERED DURING SVC 25:
        U SVC-0118 LUNO ? NOT ASSIGNED FOR I/O SUB-OPCODE ?
```

If the user enters a "?" following a display of the message, the expanded test for the I/O error will be shown. Since variable text for the I/O message is not available when the message is processed, question marks appear in the I/O portion of the message.

3.4.6 Abbreviated Forms.

If the files containing the fixed portion of the message text are not on the system disk, an abbreviated form of the message is displayed. It contains only the category, an internal message number, and any variable text that would normally have been included with the fixed text. The following examples illustrate both cases.

Assume the fixed portion of the message text for an SVC error with internal number >0027 is the following: 0315 ?1 DOES NOT EXIST. Assigning a LUNO to a nonexistent file .PRINT.OUT produces an error message as follows:

```
U   SVC-0315 .PRINT.OUT DOES NOT EXIST
```

This message appears as shown if the variable text has the value .PRINT.OUT preceded by the byte count of 10.

On a system without a fixed text file, the same message appears as follows:

```
SVC-INTERNAL CODE >0027 .PRINT.OUT
```

Suppose that a COBOL program compiles correctly and informs the user of the results. The internal message number might be >9010 and the message C06 ?1 COMPILED WITH ?2 ERRORS might be in a message file named COBOL. The displayed message appears as follows:

```
I COBOL-C06 .SOURCE COMPILED WITH 0 ERRORS
```

Without the fixed text file, it appears as follows:

```
COBOL-INTERNAL CODE >9010 .SOURCE;0
```

The variable text in this case is made up of two items, the input file pathname and the number of errors the COBOL compiler discovered. The variable text buffer is as follows:

```
9.SOURCE;0
```

where 9 is binary data indicating the length of the text string that follows.

If the utility specifies an internal message number that does not exist (for example, SVC returns >8111), the message output form is as follows:

```
SVC-UNDOCUMENTED ERROR - INTERNAL CODE >8111
```

3.5 MESSAGE FILES

Message files are maintained on the system disk to provide the fixed text portion of the messages used by SCI, the languages, and utilities. Four directories of message files are used to build and support the DNOS message facility. In the descriptions below, MESSAGES represents the volume ID of the disk that contains the DNOS source/object kit shipped by Texas Instruments Incorporated. The directories of message files are as follows:

* The MESSAGES.TEXT directory of the DNOS source directory contains the files of error and status messages in text-editable form. There is a separate file for each of the language processors, SCI, SVC processors, and the major utilities.

* The .S$MSG directory contains the information in MESSAGES.TEXT, in the format used by the operating system and SCI.

* The MESSAGES.EXPTEXT directory contains expanded explanations of the errors documented in the TEXT files. The files in this directory are in text-editable form.

* The .S$EXPMSG directory contains the information in the MESSAGES. EXPTEXT in the format used by the operating system and SCI.

Corresponding entries in the various directories have the same leaf node names. The filename SVC appears in each of the four directories MESSAGES.TEXT, MESSAGES.EXPTEXT, .S$MSG, and .S$EXPMSG, as the file of errors detected for system SVCs.

Compiler messages that appear in listings are not placed in the files; messages are centralized into one module for the compiler. Run-time error messages generally are placed into the message files. Programmers in all environments using DNOS are expected to understand English messages, but end users may not know English. Therefore, any messages displayed to end users must be easily internationalized.

All messages for a particular compiler or utility must be placed into a single file, though there may be separate files for compilers and for run-time support.

3.5.1 Details of the TEXT Files.

The TEXT message files (DNOS as well as user-defined) are editable, blank-suppressed, sequential files with a logical record length of 80 bytes. A blank line is the entry delimiter.

Record one contains the internal message number limits for the file and local language characters for the error source.

Each file in the TEXT directory has the following format:

* The lowest and highest internal message numbers are ASCII representations of the hexadecimal numbers, and are in columns 1 through 4, and 6 through 9, respectively.

* The abbreviation characters in the local language are in columns 11 through 15, in the following order:

  - User error (U)

  - System error (S)

  - Hardware error (H)

  - Warning (W)

  - Informative (I)

The remaining records of the file contain error messages in the following format:

* First line:

  - Error source indicator (U, S, H, US, UH, USH, I, or W)

  - One or more blanks

  - One or more four-digit hexadecimal internal message numbers, separated by commas and enclosed in parentheses)

  - Optional comments. They are not output with the message.

* Following Line(s). The message for this entry starts in column 1 of the next line and can be up to three lines long. This provides a maximum of 240 characters for the fixed-text portion of a message. If a message identifier is to be seen by the user, it must appear in the message.

* A blank line

A TEXT file message may contain any displayable characters with the exception of the question mark. A question mark and its associated digit are replaced with variable text when the message is processed. The question mark and digit pair may be embedded between any two valid message characters. Similarly, the ?C may be embedded between any two valid message characters.

Figure 3-1 shows a TEXT file consisting of three messages for five different internal conditions.

```
1000 10FF USHWI

U       (1000)     MESSAGE OCCURS OFTEN WITH UNPRINTABLE NAMES
FL01    FILE ?1 ALREADY PURGED

U       (1004,1014,1024)  THIS COMMENT IS NOT OUTPUT.
FL02    KEY INDEXED FILE SUPPORT NOT AVAILABLE

U       (10F0)
FN02    INSUFFICIENT BUFFER SPACE ALLOCATED IN USER TASK
```

Figure 3-1 Sample TEXT File

Assume that this file is in the S$MSG directory as a file named TESTER. A message generated for internal error code >1014 is displayed as follows:

```
U TESTER-FL02 KEY INDEXED FILE SUPPORT NOT AVAILABLE
```

Message files in the TEXT directory built by Texas Instruments Incorporated have messages in uppercase English.

The internal message numbers used within one file are independent of those used in another file. The user is free to choose any value when building a message file. It is recommended that the internal message numbers be contiguous in order to minimize file space used. The S$MSG file built from the TEXT file includes an index that is a continuous sequential table of internal message numbers and their corresponding record numbers in the S$MSG file.

If you use message files, you may want to create a companion equate file for each TEXT file of messages in use. Programs can then assemble, compile, or link with the equates module and refer to specific errors by their equated label rather than by hard-coded internal message number.

3.5.2 Details of the EXPTEXT Files.

For each TEXT file created by Texas Instruments Incorporated, there is a companion expanded explanation file in the directory MESSAGES.EXPTEXT. Files in the EXPTEXT directory are text-editable sequential files with a logical record length of 80 bytes. These files include the "Explanation:" and "User Action:" portions of the messages that appear in the DNOS Messages and Codes Reference Manual and in the information provided by the SEM command.

Files in the EXPTEXT directory created by Texas Instruments Incorporated have messages in uppercase and lowercase English.

In a file in the EXPTEXT directory, the first record contains the characters for the two phrases "Explanation:" and "User Action:" (including the colons), in the same language as the messages. The fields must be enclosed in quotes if the phrases include blanks.

Each entry for a message in the file includes the following:

* First line:

  - A key consisting of two percent-sign characters followed by the message identifier which can have as many as 14 characters. The keys %%1 and %%2 are reserved.

  - One or more blanks

  - One or more hexadecimal message numbers, separated by commas, and enclosed in parentheses

* Next line(s). One or more explaining the message.

* A blank line

* Next line(s). User action. One or more paragraphs explaining what to do when the message occurs

* A blank line

Expanded explanations may be included for some or all of the entries in the TEXT file. Figure 3-2 shows a sample EXPTEXT file corresponding to the sample TEXT file in Figure 3-1.

```
"Explanation:" "User Action:"

%%FL01 (1000)
The file indicated has already been purged with a previous
command.

Verify that the correct filename has been provided.  If not,
resubmit the command with the correct filename.

%%FL02 (1004,1014,1024)
The system in use does not have key indexed file support.  The
operations requested are not available.

Locate a system with key indexed file support or rewrite the
program to run without key indexed files.

%%FN02 (10F0)
The user task does not have enough buffer space for the
subroutine being called.

Rewrite the task providing more buffer space.
```

Figure 3-2 Sample EXPTEXT File

3.6 FILENAMES

The names of files within the four message directories may be any name except those reserved for DNOS and its utilities. Table 3-2 lists the reserved names.

Each of the reserved filenames is recognized by S$CMSG, which generates the error and status messages. It is also translated in the command procedure M$02, for use with the Show Expanded Message (SEM) Command.

To avoid conflict with the naming of other files, S$CMSG uses a file indicator rather than a filename to access the appropriate message file. The file indicator is a hexadecimal value between >01 and >FF, which is used internally by the program needing the file. The indicator is bound to the appropriate filename by the synonym $$FNxy, where xy is the indicator. While DNOS-supplied components use well known file indicators, DNOS cannot know user-defined indicators. User-defined command procedures that access user-defined message files specify $$FNxy prior to bidding a program that calls S$CMSG for error processing.

Table 3-2 Reserved Message Filenames

Filename	Use
ASSEMBLR	Assembly language completion messages
BASIC	BASIC messages
COBOL	COBOL messages
COMMON	Messages concatenated with CRASH, CRASH1 LOADER
CRASH	System crash messages
CRASH1	Messages concatenated with CRASH, COMMON
DATADICT	Data Dictionary messages
DBMS	DBMS run-time messages
DEBUGGER	SCI Debugger utility error messages
DNIO	Distributed Network I/O Messages
DNOSHLL	DNOS high-level language support messages
EDITOR	SCI Text Editor error messages
FORT78CP	FORTRAN compiler messages
FORT78RT	FORTRAN run-time messages
ICS3270	Communication messages for 3270
LINKER	Link Editor completion messages
LOADER	System loader flash crash messages
MAIL	Support for the SCI MAILBOX function
PASCAL	PASCAL run-time messages
PTP	Performance tools package messages
QUERY	QUERY run-time messages
RPG	RPGII run-time messages
S$ROUTIN	Messages common to SCI, DEBUGGER, EDITOR, UTILITY
SCI	SCI error and status messages
SMRG	Sort/Merge run-time messages
STATUS	Alias of SVC file
SVC	SVC processor error and status messages
TAP	TAP X.29 Package
TIFORM	TIFORM run-time messages
TIPE	TI Page Editor messages
UTILITY	DNOS utility program error and status messages

Table 3-3 lists the file indicators chosen by Texas Instruments Incorporated for DNOS and its utilities, for the languages, and for run-time support.

Table 3-3 Message File Indicators

Hex File Indicator	Use
00 through 2F	DNOS
01	SVC messages and codes
02	Utility messages
03	SCI messages
04	SCI Text Editor messages
05	SCI Debugger utility error messages
06	Status messages for certain SVCs
07	SVC messages and codes, in certain cases when SVC >4C was already called
08	MAILBOX support
10	Assembly language completion messages
11	System crash messages
12	System loader flash crash messages
21	Link Editor completion messages
30	3270 Communications
31	TAP X.29 package
32	Distributed Network I/O messages
34 through 3F	Reserved for Communications
40 through 5F	Language Run-Times
41	COBOL run-time support
42	FORTRAN run-time support
43	TI Pascal run-time support
44	DNOSHLL run-time support
45	RPG II run-time support
46	TIFORM executor run-time support
47	DBMS run-time support
48	QUERY run-time support
49	Sort/Merge run-time support
4A	TIPE
4B	Data Dictionary
4C	Performance Tools Package
4D-5F	Reserved
60 through 7F	Language Compilers/Interpreters
60	BASIC
61	COBOL compiler (Reserved)
62	FORTRAN-78 compiler
63	Reserved
64	Reserved
65	Reserved
66	TIFORM FDL compiler (Reserved)
67	DBMS DDL compiler (Reserved)
68-7D	Reserved
7E	TI Pascal compiler
7F	DNOSHLL compiler
80 through FF	Available to users

$$FNxy is set in the command procedure prior to bidding the program, so that the synonym is uniquely defined when message file access is required.

If the synonym $$FNxy does not exist when the message is formatted by SCI, and if the file indicator is not one of the file indicators known by SCI, the message appears with just the file indicator rather than the filename as the category identifier as follows:

```
xy--INTERNAL CODE message ID  variable text
```

3.7 UTILITIES TO BUILD THE MESSAGE FILES

Two utilities build the message files. One creates the S$MSG files from the TEXT files and the other creates the S$EXPMSG files from the EXPTEXT files. To internationalize and rebuild the entire set of DNOS messages, the batch stream .BATCH.BUILD.MESSAGE1 from the DNOS source/object kit should be used. This batch stream combines several input files to build the output files correctly.

3.7.1 Build Message File.

The SCI Build Messages File (BMF) command procedure and the BMF task build a message file in the .S$MSG directory. The BMF task consists of a Pascal program (BMF) and supporting routines. It is replicatable and nonprivileged.

BMF reads the TEXT file described above and creates two temporary files that make up the .S$MSG file. The .S$MSG output file is either a relative record file or a sequential file. The command procedure creates an output file of the specified type and merges the temporary files into it.

SCI bids BMF with a PARMS list that consists of the stack and heap sizes. Values of 1000 and 500, respectively, are sufficient. The program requires that the following synonyms be set:

$INPUT	Pathname of the input TEXT file. This file must be structured as described earlier in this section.
$INDEX	Pathname of the relative record file in which an index table (the first part of the .S$MSG file) is built.
$MSGFIL	Pathname of the sequential file in which the message file (remaining records of the .S$MSG file) is built.

The detailed formats of the output files are documented in the BMF code. The $INDEX file contains a header record (high and low internal message IDs and local language character strings) and a directory into $MSGFIL. For each internal message ID, the index table contains the record number in $MSGFIL where the associated message text is stored.

BMF opens the three files, reads the first record from the input file, and creates a complete index file. For each message, starting with the lowest message number and continuing through the highest, the index table entry (the record number of the associated text) is initialized to 0, indicating that no message text exists for the internal message code).

BMF enters a loop in which the message information is read, processed, and stored in the appropriate text record(s). Appropriate entries in the index file are updated to point to the message text.

BMF continues to process after finding an error, so that a single execution can be used to detect all errors in the input file. Errors are reported through UTPUER, the UTCOMN Pascal interface routine to DNOS message handling.

NOTE

Messages in the .S$MSG files must be displayable on VDTs, printers, and any other output device. The TEXT files and the corresponding .S$MSG files must be in all uppercase (for English). No attempt is made to translate lowercase to uppercase characters.

3.7.2 Build Expanded Message File.

A similar utility, build expanded message file (BEMF), creates the S$EXPMSG files from the EXPTEXT files. BEMF reads the EXPTEXT file, formats each explanation into a key indexed file record and writes that record to the S$EXPMSG file.

The BEMF task consists of a Pascal program (BEMF) and supporting routines. It is replicatable and nonprivileged.

SCI bids BEMF with a PARMS list as follows:

PARM	Definition
1	Pascal stack - 1500 is sufficient
2	Pascal heap - 500 is sufficient
3	$$LU - LUNO of output file
4	Convert lowercase to upper case? (YES/NO)
5	$BEMF$2 - adjusted message ID length

The program requires that the following synonyms be set:

INPUT	Pathname of the input EXPTEXT file. This file must be structured as described earlier in this section.
OUTPUT	Pathname of the error file.

Prior to bidding the BEMF task, the command procedure assigns a LUNO to the output file specified by the user. The output file may be any key indexed file, but it must reside in the S$EXPMSG directory to be used by DNOS. BEMF writes directly to the .S$EXPMSG file; no temporary files are created.

The key indexed file that is built by BEMF is capable of carrying messages with IDs of as many as 14 characters. The keys that are actually used are the two percent-sign characters followed by the one to fourteen character keys provided in the EXPTEXT file. Two special keys are built while processing the first line of the EXPTEXT file. The first string (for "Explanation:") is stored using the key %%1, and the second string (for "User Action:") is stored with the key %%2.

BEMF opens the input, output, and error files, then processes the first record to build the first record of the .S$EXPMSG file. The program enters a loop that reads and processes expanded message paragraphs until the EOF is encountered in the input file.

BEMF continues to process after finding an error, so that a single execution can be used to detect all errors in the input file. Errors are reported through UTPUER.

The key indexed file is closed prior to task termination errors. The command procedure must release the LUNO.

3.8 SHOW EXPANDED MESSAGE UTILITY

The show expanded message (SEM) utility writes the expanded text of a specified message category and ID to the specified listing file. SEM is written in Pascal and uses the UTCOMN error processing routines.

The task segment SEM is replicatable and nonprivileged.

SCI bids SEM with a PARMS list as follows:

PARM	Definition
1	Pascal stack - 1500 is sufficient
2	Pascal heap - 1000 is sufficient
3	$$LU - LUNO of message file
4	Flag to indicate whether or not to display the short form of the message: =1, means do not display short form =2, means display short form
5	Message category
6	Message ID

The program requires that the following synonyms be set:

$SEM$LST	The pathname of the listing file. If $SEM$LST has a null value, the expanded text is written to the TLF.
$$VT	Variable text
$$ES	Error source
$$MN	Internal message number

SEM calls R$CMSG, the Pascal interface routine to S$CMSG for formatting the explanation and user action portions of the expanded message.

3.9 THE MESSAGES AND CODES MANUAL

The DNOS Messages and Codes Reference Manual contains messages and explanations in a format similar to the output of the SEM command. The manual is built directly from the message files, using utilities in the messages manual data base. These utilities are described in the README file of the messages manual data base, but they are not supplied to users with either source or object versions of DNOS.

SECTION 4

SYSTEM COMMAND INTERPRETER

4.1 OVERVIEW

The System Command Interpreter (SCI) is the interface between the user and the kernel of DNOS. SCI provides service at two levels:

* The user enters an SCI primitive and keywords. SCI processes the keywords and performs the requested function. SCI issues an SVC when services of the kernel are required.

* The command procedure represents a level of removal from the primitive. In this case, SCI interprets prompts and SCI commands in the command procedure and constructs primitives and the appropriate keywords.

SCI is written in assembly language. Task structure, flow of control, and details of the routines and data structures are discussed in this section. Some comments concerning modification of SCI are included.

Refer to the DNOS System Command Interpreter (SCI) Reference Manual for details about SCI primitives, SCI command syntax, and how SCI command procedures are written.

4.2 STRUCTURE

The SCI task is made up of three segments:

* S$SYSTEM procedure segment - Library of general service routines shared by many DNOS tasks. S$SYSTEM includes only DSEG position-independent code. Any changes made to routines in S$SYSTEM must preserve this independence.

NOTE

Refer to the following sources for further information about S$SYSTEM routines:

* DNOS System Programmer's Guide
* The section of this document entitled Conventions and Libraries
* Appendix A of this document, Writing DSEG Position-Independent Code

* SCI990 procedure segment - Procedural code that performs SCI functions. This segment is shared among all executing SCIs. If changes are made to this segment, the code must remain sharable.
* SCI990 task segment - DNOS transfer vector, all volatile data for SCI (module SCITSK) and workspace, and DSEGs for S$SYSTEM routines.

The S$SYSTEM and SCI990 procedure segments are write protected.

SCIXFR is the standard DNOS task transfer vector. It must be the first module linked into the SCI990 task segment. This three word vector contains a workspace pointer, an initial program counter and an end-action address, in that order.

SCITSK is the read and write data area for SCI990. This module is linked below SCIXFR in the SCI990 task segment. SCITSK includes the following categories of data:

* Sixteen registers of initial workspace
* Return address stack (25 words)
* SVC call block structures for accessing the following entities:
  - Primary input and output devices (or files) for this session
  - Output file for a .DATA primitive
  - File in which the procedure being expanded is stored
  - Menu file

* SVC call block structures for issuing the following SVCs:
  - Map Task Name to ID SVC
  - Get Job Information SVC
  - Self ID
  - Time Delay SVC
  - Convert Binary to Decimal SVC
* Buffers
* Area for stacking as many as 32 procedure environments
* Miscellaneous data, including a 64-byte patch area

4.3 FLOW OF CONTROL

The following paragraphs describe the flow af an SCI session.

4.3.1 Invoking SCI.

The LOGON task in DNOS is responsible for initiating SCI. The LOGON task creates an interactive job and bids SCI as its initial task. This task communicates with the terminal to which it is assigned by LOGON.

Once SCI is active, it can initiate SCI as a background batch task within the same job. This background task has access to resources through its parent task, the SCI task that bid it.

SCI can create a batch job in which SCI is a task. Once this batch job has been created, all ties with the parent job and task are severed. Resources are not shared between the batch job and the parent job.

4.3.2 Initialization.

Initialization of SCI consists of the following:

* Gaining access to input and output resources, synonyms, and logical names
* Determining the mode (batch or interactive) in which SCI is functioning

* If interactive mode, getting terminal information from the TINFO task

* Establishing communication with MAILBOX

* Showing the news file .S$NEWS when appropriate

* Invoking the command procedure M$00.

SCI invokes M$00 to allow a user to perform one or more operations deemed desirable at the beginning of every SCI session. For example, the command procedure can be used to customize prompts, menus, and command libraries. A companion command procedure, M$01, is invoked at the termination of an SCI session.

4.3.3 Major Loop.

Following successful completion of the initialization process, SCI enters its major processing loop, which performs two functions:

1. Displays the terminal local file (TLF), menus (if they exist) and messages (if they exist)

2. Gets and processes the next input

Unless end-action is taken by SCI or the user specifically terminates SCI, control returns to this major loop following completion of command processing, or any time the Command key (CMD) is pressed.

4.3.4 Termination.

The SCI990 task terminates during processing of the .STOP primitive. This primitive is in the Quit (Q) command procedure for an interactive session and in the End Batch Execution (EBATCH) command procedure for a batch session.

4.4 DESIGN CONCEPTS

The following paragraphs describe some overall concepts of the design of SCI.

4.4.1 Command Procedures.

In command mode, SCI recognizes two kinds of user input -- an SCI language primitive or the name of a command procedure. A command procedure is a collection of SCI primitives and/or other command procedures.

A procedure may invoke other command procedures, but ultimately, when all nested procedures have been expanded, the result is a series of one or more primitives. To expand a command procedure is to read nested command procedures until all procedure references have been resolved to SCI primitives.

SCI maintains a variable (DEPTH) that is a measure of the current command procedure nesting depth. Depth level 0 is the primary input level (that is, the batch input file or the interactive terminal). DEPTH is incremented and decremented as command procedures are entered and exited. The maximum nesting level is 32. SCI processes each command line using field prompts defined in that command procedure, at that nesting depth. SCI commands can be called recursively.

4.4.2 Environment Stacking in Nested Procedures.

SCI stacks the following elements of the environment when a new nesting level is entered:

* LUNO for the current command procedure

* Record number in the current procedure

* IF/LOOP counter. Since .IF and .LOOP structures must be terminated (by .ENDIF and .REPEAT, respectively) before the end of the command procedure, this count is maintained at each level of nesting.

* Expert mode flag. The variable EXPERT. Any nonzero value implies that expert mode is active.

* Stage depth. Not currently implemented

* The name of the command procedure

The environment information requires 17 bytes of memory per nesting level.

In effect, the field prompt values are stacked, because of the way they are stored by SCI. When a field prompt response is put into the name correspondence table (NCT), SCI pairs the value with a name that has the following format:

```
00<run ID><depth level><name>
```

where:

run ID	is a binary number that is the run ID of the parent task.
depth level	is the current value of DEPTH (a binary number). The depth level >50 is a special value used for temporary storage of field prompt/value pairs entered in expert mode. (See the discussion of XPROMP for details.)
name	is the name of the field prompt.

SCI appends the same information to a field prompt name when requesting the value from the Name Manager. The only field prompt values that are available to a command procedure are those defined at that command procedure's depth level.

The environment is unstacked when the end of a procedure is reached. When the procedure depth level is popped, all name/value pairs at the depth being exited are purged from the NCT.

4.4.3 Task Bidding.

Four SCI primitives are used to bid tasks. They are as follows:

* .BID activates the specified task as a foreground activity and suspends SCI until the task terminates.

* .QBID activates the specified task as a background activity in a new stage. SCI suspends in a batch session, but does not suspend in an interactive session. Tasks activated by .QBID do not share synonyms and logical names with SCI.

* .DBID activates the specified task in a new stage, with the specification that the task is to be suspended immediately. This primitive is for the Debugger utility.

* .RBID activates the specified task and suspends SCI. Tasks activated by .RBID can return control to SCI without terminating.

Table 4-1 is a summary of two characteristics of each type bid: whether or not SCI is suspended, and whether or not the synonyms and logical names are shared by SCI and the task that is bid.

Table 4-1 Task Bid Characteristics

Interactive / / Batch	.BID	.QBID	.DBID	.RBID
Suspend SCI?	Yes / Yes	No / Yes	No / (Note 1)	Yes / Yes
Share Synonyms and Logical Names?	Yes / Yes	No / No (Note 2)	No / (Note 1)	Yes / Yes

Note 1 - Not allowed in batch mode

Note 2 - Incompatible with DX10

In processing each of the primitives, an Execute Task SVC (>2B) is issued when the task is first bid. Refer to the <u>DNOS SVC Reference Manual</u> for details of the call block. Four flags in the flags byte (byte 3 of the call block) vary with the four bid primitives, and with interactive or batch mode. Table 4-2 shows the flag states.

Table 4-2 Flag States for >2B SVC Call Block

Interactive / / Batch	.BID	.QBID	.DBID	.RBID
RBID task (Bit 2)	0/0	0/0	0/*	1/1
Background task (Bit 3)	0/0	1/1	1/*	0/0
Unconditional suspend (Bit 6)	0/0	0/0	1/*	0/0
Suspend calling task (Bit 7)	1/1	0/1	0/*	1/1

* Not allowed in batch mode

The RBID concept is implemented to allow alternating execution between SCI and utility tasks, such as the Text Editor and the System Configuration Utility. By calling the S$WAIT routine, a utility task can relinquish control to SCI and remain in the user's job until reactivated by SCI. The SCI routine S$RBID and the S$SYSTEM routines S$NEW and S$WAIT coordinate the switching of control between SCI and an RBID task. They use the Execute Task SVC, the Unconditional Suspend SVC, the Activate Suspended Task SVC, and the synonym $$RBID.

SCI maintains a table of active RBID tasks, in the SCI990 task space. The name of the structure is RBIDAC. It contains a maximum of five two-byte entries. Corresponding to each entry in the list is a task in the user's job that has been bid using the .RBID primitive, and has returned to SCI by calling S$WAIT. The entry in this table is the installed ID of the task and the run ID of the task in the user's job.

The routines S$RBID, S$TERM and S$WAIT use the synonym $$RBID. When SCI passes control to an RBID task, a non-null value indicates that SCI is taking end action. The utility task takes the appropriate action, that is, it should terminate. This is the last opportunity for the .RBID task to do any cleanup processing and to call S$TERM.

When the utility task passes control back to SCI, a non-null value for $$RBID indicates that the utility task was suspended by S$WAIT and should remain in the active RBID task list (RBIDAC).

WARNING

.RBID is intended for the exclusive use of SCI and DNOS utilities. Texas Instruments, Inc. does not guarantee implementation detail consistency in future releases of DNOS. Any application programs written to exploit this feature may fail in future releases.

4.4.4 SCI Subroutine Linkage.

A push/pop stack of return addresses is maintained for executing subroutine calls and returns in SCI code. The SCI CALL macro generates a branch to SDCALL, whose address is always maintained in register 10. SDCALL checks for and reports stack overflow. A maximum of 25 nested calls can be stacked.

The return routine has two entry points -- SDSRET and SDSERR. SDSRET unstacks the call and returns control to the calling routine at the instruction following the CALL macro.

SDSERR is the error return entry point that reports an error. The call is unstacked and control is transferred to the error address specified on the CALL macro. If no error address is specified, control is transferred back to the calling routine at the location following the CALL macro.

No working registers are saved or restored with a normal call/return sequence within SCI. The routines SAVE09 and RSTR09 are available to save and restore registers 0 through 9. The buffer for storing registers is twenty bytes in length, a stacking facility for only one level.

4.4.5 Macros.

The macros required by SCI are in the file DSC.SCI990.SOURCE. MACROS. The macros are as follows:

LTXT Produces a data structure consisting of the appropriate byte length, followed by the character string that is the operand of LTXT.

ZTXT Produces a data structure consisting of the appropriate byte length, followed by two bytes of zero, followed by the character string that is the operand of ZTXT. (ZTXT produces a structure with the same format as a field prompt name in the NCT.)

SEC Produces the appropriate calling sequence for S$XFER

CALL Produces a branch and link sequence to the first operand with an optional error exit to the second operand. This macro implements the CALL portion of the SCI subroutine linkage strategy. The format of the macro is as follows:

```
CALL   @ROUTINE,ERROR
```

where:

ROUTINE	is the routine to be called.
ERROR	is the address at which an error encountered by ROUTINE is processed.

The following instructions are generated:

```
BL          *R10   (R10 = address of SDCALL)
DATA        ROUTINE,ERROR
```

4.4.6 Error Reporting.

The error reporting system in SCI consists of two phases:

1. When an error occurs, the information required to format the proper message is saved.

2. The message is displayed on an interactive terminal or in the batch listing file.

The standard error reporting interface, for both SCI and any tasks it bids, is S$TERM. This routine sets the termination synonyms and terminates any task bid by SCI. S$TERM treats a call from SCI as a special case in which the calling task is not terminated. Register assignments for S$TERM are covered in detail in the DNOS Systems Programmer's Guide.

SCI reports one error for each command input by the user. The routine that first encounters an error condition does not always have the information required to determine which message or action is most useful to the caller and/or user. The routine S$XFER provides some flexibility in reporting errors.

When a utility routine detects an error, it uses the set error condition (SEC) macro which branches to SDSERR, which calls S$XFER.

S$XFER buffers error reporting in order to minimize the number of calls to S$TERM. This buffering of error conditions is an important performance consideration. Considerable overhead is involved when S$TERM calls the Name Manager to set termination synonyms.

Nested routines may ignore or recover from certain error conditions. A code is passed to S$XFER to control the state of the buffered error. The meaning of the code values are as follows:

Code	Action
1	Sets the error condition variables and calls S$TERM. In this case, any previously held condition is passed to S$TERM. This allows a calling routine to report the error encountered by a called routine.
2	Resets the error condition. This causes all previously held error conditions to be cleared. This allows a calling routine to nullify any error reported by called routine(s).
3	Holds the error condition. This causes future error conditions to be ignored until a set or reset request is processed.
4	Terminates error reporting. This is a special case for processing the .STOP primitive in batch SCI. After calling S$TERM, the TEXT and CODE keyword values are used to set the termination synonyms $$VT and $$CC, respectively. All future errors are ignored. (The only errors that should be encountered are in the LOGOFF procedure M$01.)

The phrase termination synonyms is used to refer to the following set of synonyms:

* $$CC (Condition Code)

* $$VT (Variable Text)

* $$ES (Error Source)

* $$MN (Message Number)

* $$FN (Filename)

* $$PN (Additional Text: procedure name and line number)

These synonyms (with the exception of $$PN) are set by S$TERM. S$XFER sets $$PN. The termination synonyms are used by DERROR to format the message, if any, to be displayed.

Routines called in the display step of the major loop report errors through DERROR, which calls S$TERM directly rather than through S$XFER. Any error encountered in these routines is reported to the user.

DERROR is designed to display messages in the following order:

1. Messages to report errors flagged by the foreground task

2. Messages to report errors flagged by one or more background tasks

3. Messages from MAILBOX

Foreground error messages are produced according to the values of the termination synonyms. Background error messages are produced by examining the values in descendant error lists (DELs). These values correspond to the values of termination synonyms set in descendant stages by S$TERM. DELs are data structures implemented by the Name Manager. See the <u>DNOS System Design Document</u> for further details.

In the current release of DNOS, only one background task is allowed (and therefore, no more than one DEL is produced). Should this limit be removed, DERROR is designed to process DELs from multiple background tasks. The algorithm implemented is a loop that consists of reading a DEL and displaying the message, until the last DEL is processed.

S$CMSG formats error messages. The required register assignments and the formatting process are documented in the <u>DNOS Systems Programmer's Guide</u>.

Termination messages from batch SCI jobs will be logged to the system log file and listing file. In the event the listing file does not exist, the message will be written only to the system log. No message will be written to the system log for normal termination.

4.4.6.1 SCIERR.

Using the DEF and EQU facilities of the assembler, this module establishes a label for each error recognized by SCI. A label is used instead of a hexadecimal constant when an error is reported. Under this scheme, an assembly cross-reference listing summarizes all routines that generate the error of interest.

The existence of SCIERR also provides a summary listing of all errors SCI reports. Whenever a new error condition is added to SCI, SCIERR is extended and the appropriate message files are updated. See the section of this document entitled Error and Status Message Handling for details of the message files.

4.5 DETAILED DESIGN

SCI can be divided into the following functional groups of code:

* High-level routines - Main driver, major loop processing, and SCI constant data declarations

* Command procedure processing routines - Control the flow of information between command procedures at various levels

* Primitive processing routines - Process SCI language primitives

* Parsing routines - Set up and perform textual substitution on the command buffer and search for specific entities

* Display routines - Process data to be displayed (messages, menus, etc.) and write it to the appropriate file or device

* Subsystem support routines - Interface with MAILBOX, the subsystem that distributes messages, and with TINFO, the subsystem that maintains terminal information

* Utility routines - Perform basic functions. They are used by all functional groups of code.

4.5.1 High-Level Routines and Modules.

The high-level routines shown in the call tree Figure 4-1 are described in detail in the following paragraphs.

```
                             SCI990
                             --+---
                               |
                +--------------+--------------+
                |                             |
              DMENU                         GETCMD
         +-----------+           +------------+----------+
         |           |           |            |          |
      DERROR     LIBSCN        GETOPC       XPROMP/     XSTOP
         |  (GETMNU Entry)                  EXPROC
         |                                    |
      GETCMD                                LIBSCN
  (GETLCM Entry)
```

Figure 4-1 Call Tree for SCI High-Level Routines

4.5.1.1 SCI990.

The main driver for SCI is SCI990. This routine calls for all required initialization and executes the major loop.

SCI990 operates in one of three states, depending on I/O requirements. Internally, the state is indicated by the value of the one-word global flag STATE.

State	STATE Flag	I/O Requirements
Batch	0	Input from a sequential file or device Output to a different file or device
TTY	>0001	Input from an interactive device that is not a VDT Output to the same device
VDT	>FFFF	Input from a VDT keyboard Output to a VDT screen

The following four bytes of information are available to the SCI through the Get Task Parameters SVC:

* Byte 0. Eight bits of terminal status information, as follows:

  - Bit 0. Reserved. Always zero

  - Bits 1 through 3. Privilege level of the user

  - Bits 4 through 7. Terminal mode. This is a four-bit representation of the proper value of the STATE flag.

* Byte 1. Station number of the physical terminal with which this task is affiliated. When the station number is in the range >01 through >FE, this is a background SCI task, and any terminal information available is in byte 0. The station number values of >00 and >FF have the following special meanings:

  - >00. This is an interactive task. Terminal information is not included in byte 0.

  - >FF. This is affiliated with a batch job and has no access to a terminal.

* Byte 2. Value of the optional parameter CODE on the task bid that invoked SCI. This information is not

presently used by SCI.

* Byte 3. Zero for SCI.

Note that when SCI is invoked as an interactive foreground task by LOGON, no terminal information is passed in byte 1. A zero value for station number implies that the information must be obtained from other sources.

Terminal mode is obtained from the TINFO subsystem. Privilege level is obtained through a Job Manager SVC, and station number through a Self Identification SVC.

4.5.1.2 DMENU.

DMENU displays the TLF, defines certain synonyms, calls DERROR to display messages, and, optionally, displays a menu.

The following synonyms are set when the user bids SCI and cannot be changed.

* $$MO. Mode of the session:
  - >00 Batch
  - >01 TTY interactive
  - >0F VDT
* $$ST. Two-digit decimal station number for the interactive session. This synonym has a value of zero for a batch job session.
* $$UI. User ID with as many as eight characters
* $$12. Yes/no flag for the existence of 990/12 hardware
* ME. Four-character station name. This synonym is deleted in a batch job session.

4.5.1.3 DERROR.

DERROR calls S$CMSG to format S$TERM, S$STOP, DEL, and MAILBOX message information. DERROR writes the information to an interactive terminal or to the batch listing file.

Once the foreground message has been displayed, DERROR resets bit 5 in the synonym $$ES. This prevents the message from being displayed again. In interactive VDT mode, DERROR examines the next user input. If it is a question mark (?), command procedure M$02 is invoked. This command procedure displays the expanded message text to the user.

In batch mode, the value of the synonym $$CC is examined. If it is greater than zero, an 80-character highlight line of the following format is written to the batch listing file:

```
<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>*<>
```

For batch SCI jobs, the final termination messages will be logged to the listing file and the system log file. If the listing file does not exist the message will be written only to the system log. No message will be written to the system log file for normal termination.

Message contents and files, as well as formatting rules and examples, are described in detail in the section of this manual entitled Error and Status Message Handling.

4.5.1.4 GETCMD.

Routine GETCMD is responsible for reading a command line, identifying the desired function, and transferring control to the appropriate processing routine. GETCMD identifies three types of functions:

* Command line is a primitive
* Command line invokes a command procedure
* Command line is the end of a command procedure

GETCMD calls GETLNE to read the input line into the command buffer and calls TXTSUB to make substitutions for synonyms and field prompts. GETOPC is called to parse the operation code specified in the command buffer.

GETOPC returns a result in a register. If the input is a primitive, the leftmost byte is zero, and the rightmost byte is an index into a table containing addresses of routines that process the SCI primitives.

The address table in GETOPC offers a convenient way to disable an SCI primitive. A check is made to detect the loading of a value of zero into the branch register. If zero is loaded, the branch is not taken, and an error message is generated. In the current release of DNOS, the following primitives are disabled in this manner: .STAGE, .EOS, and .COPY.

If the leftmost byte of register one is nonzero, the register contains a pointer to a buffer containing the name of the command procedure that is to be expanded. GETCMD calls EXPROC to do this expansion.

4.5.1.5 GETOPC.

GETOPC has responsibility for determining whether the command is a primitive or a procedure call. This determination is made by examining the first nonblank character in the command buffer. If it is a period, the command is processed as a primitive.

GETOPC identifies the primitive by searching a table of primitive names. The character strings that are names of primitives are stored in the name table in the same order as the addresses of routines that process the primitives are stored in the branch table. The index into the name table is the index into the processing routine address table. GETOPC returns this index. If no match is found in the name table, GETOPC generates an error.

The user request that a menu be displayed is a special case in GETOPC. When the first nonblank character of the command buffer is /, that slash is replaced with the character string .MENU. For example, if the user enters /EDIT, GETOPC converts that string to .MENU EDIT. This feature serves two functions:

* Provides a shorthand way to request the menu for any command procedure

* Permits a user who is not normally allowed to enter primitives from the terminal to, in effect, enter the primitive requesting menus.

If the command is not a primitive, GETOPC returns a pointer to the buffer that contains the name of the procedure.

4.5.1.6 LIBSCN.

LIBSCN directs access to command procedures and menus defined in the primary and secondary libraries.

The entry point to gain or release access to a menu is GETMNU. The entry point to gain or release access to a command procedure is GETFIL.

Stacking or unstacking the command procedure environment is done when LIBSCN is called through the entry point GETFIL.

When a new command procedure is invoked, LIBSCN stacks the current environment, opens the new input file for reads, and increments the procedure depth counter.

When access to a command procedure is released, LIBSCN closes the current input file, unstacks the previous command procedure environment, and decrements the procedure depth counter.

When LIBSCN is called to push the procedure environment, it is passed a pointer to the name of the new input source. The name supplied is appended to the name of the primary procedure library directory (the library name is kept internally in the variable USYS), and the LUNO for input resource is assigned to the new file from which input is to be obtained. The file is opened. If the assign and open produce no errors, LIBSCN processing is complete.

If an error other than >27 (pathname undefined) or >21 (volume not installed) is returned by the assign, the error is reported and processing in LIBSCN terminates.

If the command procedure is not found in the primary library, LIBSCN attempts to locate it in the secondary library. The specified name is appended to the secondary directory name pointed to by USE. The LUNO is assigned and the file is opened. All errors are reported.

4.5.1.7 XSTOP.

XSTOP processes the .STOP primitive, which terminates SCI. The processing is different, depending on whether the session is interactive or batch.

Interactive Session.

The following processing is done by XSTOP when the .STOP primitive is from an interactive session:

1. Ensures that no background tasks or RBID tasks are active. If so, an error is generated and the .STOP is ignored.

2. Invokes M$01.

3. Calls DERROR to process messages and terminate communication with MAILBOX.

4. Performs processing in routine WRAPUP.

   a. Aborts RBID tasks. (This aborts anything RBID in M$01.)

   b. Copies the current synonyms and logical names to the permanent files. The pathnames of these files must be .S$USER.userID.SYN for the synonym file and .S$USER.userID.LGN for the logical name file. In both pathnames, userID is the ID used to initiate the SCI session.

   c. Clears the VDT screen, if VDT state.

   d. Closes the input and output files and/or devices.

e. Deletes the foreground and background TLFs.

f. Issues an SVC to terminate the task.

Batch Job Session.

The following processing is done when by XSTOP when the .STOP primitive is from a batch job session:

1. Copies the TLF to the listing file.

2. Processes the TEXT and CODE parameters if they are specified with the .STOP. Sets the termination synonyms $$VT and $$CC to TEXT and CODE values, respectively.

3. Invokes M$01.

4. Writes the M$01 TLF to the listing file.

5. Performs processing in routine WRAPUP.

   a. Aborts RBID tasks.

   b. Closes the input and output files and/or devices.

   c. Deletes the TLF, if this is a batch job.

   d. Issues an SVC to terminate the task.

4.5.2 Command Procedure Processing Routines.

All user inputs that are not primitives are processed in module XPROMP. The routines discussed in the following paragraphs initiate processing of the command procedure and ensure that user-provided inputs have valid characteristics, as defined in the command procedure.

Field prompts may be defined either on the same line as the command procedure name or on a .PROMPT primitive. The information for each field prompt -- name, initial value, whether or not a value is required, and acceptable data type(s) for the value of the prompt -- satisfies the same syntax rules regardless of where the definition appears.

A large part of the processing in these two instances is implemented by common code in the module XPROMP. Initialization for processing a newly invoked command procedure is done at entry point EXPROC and for a .PROMPT primitive at entry point XPROMP.

Flow through the code in XPROMP is complex and not always obvious. The logic sketches in the discussion of common code are intended solely to convey logic. The implementation contains more GO TO transfers and common paths that sometimes contain redundant testing.

4.5.2.1 Entry Point EXPROC.

The code at entry point EXPROC initiates the processing of a command procedure. Upon entry, the name of the desired procedure is stored in the buffer PROCNM. The command buffer pointer CBPTR points to the first nonblank character following the name of the procedure.

EXPROC initialization consists of the following:

1. Calling the Name Manager (through routine S$PCNT) to purge all field prompt names and values at expert depth in the NCT.

2. Calling GETEOL to determine whether any nonblank characters appear after the command procedure name. (GETEOL sets CBPTR to point to the next nonblank character when that character is not the end-of-line byte.) If there is information on the command line, the user is in expert mode and the flag is set by EXPROC. Routine KEYLST is called to store all name and value pairs specified on the command line. KEYLST strips enclosing parentheses from lists. No verification of values is done. At this point, the names and values are associated pairs stored at expert depth in the NCT. If a name appears in the assertive state (no value assignment), it is paired with a value of Y.

3. If this is batch mode, setting the expert mode flag. There is no resource that can be prompted for values, so it is assumed that all required information is supplied on the command line. Even if no information is specified, an attempt is made to execute the command procedure using initial values of field prompts.

4. Calling LIBSCN, through entry point GETFIL, to gain access to the specified command procedure.

5. Checking the first line of the procedure to ensure that this is the command procedure desired. Unless the first line of the procedure begins with name or .PROC name, where name matches the character string in the buffer PROCNM, processing is aborted and an error is reported.

6. Initializing PRMTLN to a value of zero to indicate to

common code that entry was through the entry point EXPROC.

Following this initialization, control passes to common code in module XPROMP.

4.5.2.2 Entry Point XPROMP.

This entry point is used when a .PROMPT primitive is processed. Initialization consists of two operations:

* If DEPTH=0, aborting processing. The .PROMPT primitive is not accepted from the interactive terminal or from the batch input listing file.

* Initializing PRMTLN to a value of 1 to indicate to common code that entry was through the entry point XPROMP.

4.5.2.3 Common Code for EXPROC and XPROMP.

The code that is common to entry points EXPROC and XPROMP is in module XPROMP.

The output of the common code is a set of name/value pairs stored at the proper depth in the NCT. Each of these values has attributes that satisfy the declarations in the procedure being processed.

Command processing in this common code is in one of four modes: normal mode, ENTER key mode, expert mode, or expedite mode.

* Normal mode is the absence of the requisite conditions for any other mode -- that is, it is not expert mode, not enter key mode and not expedite mode. In normal mode, the screen is formatted and the user is allowed to supply values for all field prompts.

* ENTER key mode is established when the user in normal mode presses the ENTER key. This mode simulates a carriage return as the user response to every remaining field prompt. The ENTER key mode is reset when a subsequent field prompt does not have an acceptable value. Normal mode is reestablished.

* Expert mode is activated when one or more characters appear on the command line beyond the name of the command procedure. Expert mode remains active for all prompting within the command procedure. Expert mode is an attribute of the environment at each depth, and is stacked. Expert mode reduces the number of reads from the interactive terminal, and minimizes the processing

required to validate data from those reads.

In expert mode, the user is allowed to supply input only at depth level one, and only for a field prompt that does not have an acceptable value. In this case, any field prompt prior to the one for which the user is prompted may be changed, but expert mode is not reset.

Expert mode does not transfer to procedures invoked by the one being processed in expert mode. For example, assume that procedure A is invoked in expert mode and procedure A invokes procedure B. Whether or not procedure B is processed in expert mode is determined by how procedure A invokes procedure B (with or without one or more nonblank characters past the name of the procedure). In a batch session, all commands are processed in expert mode.

* Expedite mode is a submode of expert mode and is active as long as the predicted number of reads to the interactive terminal is zero. No terminal reads are required if all field prompt conditions are satisfied in expert mode. If the command procedure is invoked in expert mode, the attempt is made to process the command procedure in expedite mode. Expedite mode is reset when a required field prompt has no acceptable value. The performance advantages of expedite mode are realized if all required field prompts have values that are verified before the user is prompted. These values are either initial values or values supplied by the user in expert mode.

XPROMP is called to process each .PROMPT primitive in a command procedure. It is possible that one part of a command procedure qualifies for expedite mode and another does not qualify.

In this discussion, the following terms are used:

* Initial value - Value supplied in the definition of a field prompt or by the user, in expert mode. (These are called DUMMY values in code comments.)

* Actual value - Value supplied by the user in response to a prompt at the interactive terminal.

The following variables are used throughout XPROMP:

* EXPERT - Flag that reflects the status of expert mode

* XPDITE - Flag that reflects the status of expedite mode

* ARGDSP - Index equal to two times the number of field

prompts declared in the command procedure. This variable is always incremented and decremented by two because it is used as an offset into tables that have two-byte entries. ARGDSP has a maximum value of 44. (As many as 22 field prompts can be declared.)

* TYPTBL - Index into field prompt data structures

* KWTBL - Data structure containing pointers to field prompt names

* VALTBL - Data structure containing pointers to field prompt values

* ENTERK - Flag that reflects the status of ENTER key mode.

* CURPOS - Flag for S$GKEY. When set, this flag indicates that the horizontal cursor position (column) is not to be changed.

* ERRPTR - Pointer to an error message that is to be displayed by S$GKEY. A value of zero indicates no error is to be displayed.

* PRMTLN - Flag that indicates which entry point (EXPROC or XPROMP) is used to enter XPROMP. If EXPROC was used name/value pairs are stored at expert depth, and no name/value pairs are stored at the current depth.

Three functional sections of common code exist. In the first section, the field prompt definitions are processed. Any initial value (either supplied on the statement that defines the prompt or supplied by the user in expert mode) is examined for appropriate characteristics. If necessary, the screen is formatted.

The second section of processing in common code is a loop that verifies the attributes of the value for each field prompt from the interactive terminal. This loop is exited only when acceptable values are available for all required field prompts, the CMD key is pressed, or an irrecoverable error occurs.

The final section writes a message to the user, when appropriate.

Certain conditions are examined each time it is possible to take one of the following shortcuts:

* Avoid formatting the screen. This saves I/O to the screen.

* Avoid prompting the user for further input. If all information known to be required at this point in the

command procedure is available, the values are verified in section one, and section two (including reads to the terminal, verification processing, and Name Manager overhead) is bypassed.

Processing Field Prompts.

When common code is entered, the command buffer contains one or the other of the following:

* The name line of the command procedure, with CBPTR pointing to the first nonblank character

* The .PROMPT line, with CBPTR pointing to the first nonblank character after .PROMPT

The command buffer is parsed for full name and/or privilege level information. The full name is stored in FNBUFF and the privilege level is checked against the privilege level of the user invoking the procedure. Processing is aborted if the user's privilege level is lower than that specified in the command procedure.

Initial values are processed by the routine DUMARG. For each field prompt, GETALT is called to build the table of characteristics that a value for the field prompt must satisfy. In DUMARG, if expedite mode is active, an attempt is made to verify the initial values and maintain expedite mode. An initial value supplied by the user in expert mode overrides an initial value defined in the command procedure. If expedite mode is not active, verification is not performed until the next section of the code, in which field prompt values are verified.

Following the processing of field prompt names and any initial values specified, the interactive screen is formatted, except in the following cases:

* Batch mode

* Expert mode at a depth level greater than one (that is, in nested procedures)

The screen is formatted at depth level one, even though there may be no opportunity for the user to enter values for field prompts. When the screen is formatted in expert mode, the full name buffer is blanked out to indicate to the user that this is expert mode.

At the end of section one, KWTBL and VALTBL contain pointers to field prompt names and pointers to any values that exist (initial values).

Verifying Field Prompt Values.

The second section of common code contains a loop that verifies the values supplied for each field prompt, starting with the first. If the expedite mode flag is set at the end of the first section, every required field prompt has a value that is already verified. In this case, the second section is bypassed and wrap-up processing begins.

The variable ERRPTR is cleared.

The logic of the loop is as follows:

```
LOOP:  For each field prompt;
 IF expert mode is set
    THEN
       IF a value is available in VALTBL
          THEN Enter carriage return processing
             at ALTERNATE ENTRY;                      (Note 1)
          ELSE
             IF the field prompt is optional
                 THEN GO TO LOOP;
                 ELSE
                      Reset ENTERK;
                      IF batch mode
                         THEN Exit LOOP with error;
                         ELSE GO TO label PROMPT;
    ELSE
      PROMPT: IF DEPTH is not 1
                 THEN
                    IF expert mode is set              (Note 2)
                       THEN Exit LOOP with error;
              IF the ENTERK flag is set                (Note 3)
                  THEN Process this as a carriage
                       return;
                  ELSE
                       Call S$GKEY to get a value;
                       Process according to event
                        character;                     (Note 1)
END LOOP;
```

==

Note 1 - Processing of event characters is described in subsequent paragraphs.

Note 2 - This test is required because it is possible to branch from the THEN clause into the ELSE clause (at the label PROMPT).

Note 3 - At this point, the screen is known to be formatted.

Event characters are processed inside the loop shown in the immediately preceding metacode. The processing of each event key is as follows:

* Command (CMD) Key. Returns to the primary input source. The routine XEOP is called and the depth counter (DEPTH) is decremented until DEPTH=0. Control is returned to the major loop of SCI.

* ERASE INPUT Key. Starts over on field prompt values. The values for field prompts are reset to the values established in the first section. Control is then returned to the end of the first section of common code, where the screen is (re)formatted. The verify field prompts section of code is reentered. This can be done because the values set by DUMARG have been verified and are in the NCT at the current depth level. Any values specified in response to prompts have only been stored in VALTBL (the Name Manager has not been called to store values in the NCT).

* Down Arrow. Sets the flag CURPOS for the next S$GKEY call. This keystroke is processed as a carriage return except when it is entered on the line containing the last field prompt. On the last line it is treated as a no-op.

* ENTER Key. The flag ENTERK is set to indicate that this key has been pressed and the ENTER key mode has been activated. Control is transferred to carriage return processing.

* Carriage Return:

  - NORMAL ENTRY. Carriage return processing assumes that the user has specified a value for a field prompt. SQUISH (an internal routine) is called to remove leading blanks and quotes surrounding the text supplied by the user.

  - ALTERNATE ENTRY. The command buffer is set up to appear as though the value had been supplied on the command line. GETVER is called to determine if the value supplied meets command procedure specifications for the field prompt. After return from GETVER, the following logic is executed:

```
          IF the value acceptable
              THEN Put the value in VALTBL;
              ELSE
                Set ERRPTR for display of error message;
                IF batch mode
                   THEN Exit LOOP with an error;
                IF DEPTH = 1 and not expert mode
                    THEN
                       Clear the ENTERK flag;
                       IF batch mode
                          THEN Exit LOOP with error;
                       GO TO label PROMPT in LOOP;
                    ELSE Exit LOOP with error;
          ENDIF;
```

Once LOOP has been exited, S$KEY is called to place each name and value pair into the NCT at the current depth.

Writing a Message to the User.

S$WIT is called to write the following message (in the local language) to the interactive screen:

FOREGROUND COMMAND EXECUTING

This call to S$WIT is bypassed only in the following cases:

* Batch mode

* TTY mode

* Expert mode at a depth level greater than one (that is, in nested procedures)

4.5.2.4 DUMARG.

This routine processes field prompt type specifications and builds the KWTBL and VALTBL data structures. KWTBL contains pointers to field prompt names to be formatted on the screen. VALTBL contains pointers to values for field prompts. (The value may be null.)

Expedite mode is controlled by DUMARG.

Once DUMARG has built an entry in KWTBL and VALTBL, S$KEY is called to store the name/value pair in the NCT at the current depth. If a name/value pair has no declaration (the type declaration may appear on a .PROMPT statement later in the procedure), the pair is still stored at the proper depth, and verification is done when the type declaration is available.

Initialization in DUMARG consists of clearing KWTBL, VALTBL and ARGDSP.

The loop in DUMARG implements the following functions:

* Gets the name of the field prompt and stores it in KWTBL.

* Calls GETALT to build an entry in a buffer, according to the specified type declarations.

* If in expert mode, calls S$MAPK to determine whether a value for this field prompt is specified on the command line. If so, deletes the name/value pair at expert depth and saves the value in a temporary buffer for later use.

* Gathers information to be used in formatting the screen, should it be necessary. This involves examining the length of each field prompt name. A limit of 28 characters is imposed on the displayable width of a field prompt name. If the name is longer than 28 characters, it is truncated. The screen is formatted, leaving the maximum possible number of blank columns to the right of the field prompt names.

* The following special case for initial value is processed: If the initial value of a field prompt begins with the character $, a null string is used as the initial value. This is implemented because it is common to use synonyms as initial values of field prompts. If the synonym is undefined, routine TXTSUB substitutes the character string itself in the text of the command buffer. Without special processing, this value would be paired with the field prompt name as its value. Since field prompt name/value pairs are displayed and are often passed to tasks , the null string is considered more indicative of the undefined status of the field prompt value. This convention is enforced in code, and any field prompt initial value that begins with the character $ is replaced with the null string.

When the following logic is entered, the command buffer contains the first line of the command procedure or the .PROMPT line. Any field prompt values supplied by the user in expert mode have been stored in the NCT at expert depth if common code is entered through EXPROC, and at the current depth if through XPROMP.

```
  LOOP: For all field prompt declarations
     Get field prompt name and save in KEYWRD;
     IF expert mode is active
        THEN
           IF EXPROC entry point
              THEN
                 Save current depth;
                 Set expert depth;
                 Call S$MAPK for a value;
                 IF a value is returned
                    THEN Delete name/value pair at
                       expert depth;
                 Restore current depth level;
        ELSE Call S$MAPK for a value;
     ENDIF;
(Note 1)
     IF the next character is =
        THEN
           Call GETALT; (build AUX2 entries for field prompt)
           Call GNB; (advance CBPTR to next nonblank character)
           IF there is an initial value
              THEN
                 Parse initial value;
                 IF value starts with $
                    THEN VALUE = null string;
                    ELSE VALUE = value;
                 IF anything is stored in AUXBUF
                    THEN VALUE = AUXBUF;
              ELSE VALUE = AUXBUF;
           Update KWBUFW for length of name in KEYWRD;
           Store pointers to name and VALUE in KWTBL and
            VALTBL, respectively;
           IF expedite mode
                 THEN
                    Call GETVER to verify VALUE;
                    IF VALUE is not acceptable
                       THEN reset expedite mode;
     ENDIF;
     Call S$KEY to store name/value pair in the NCT at current depth;
  END LOOP;
```

Note 1 - At this point S$MAPK has stored some value for the field prompt in the temporary buffer AUXBUF. Any field prompt with no value is paired with the null string.

The final section of DUMARG writes the field prompt names and values to the batch listing file, if appropriate. If any field prompts remain at expert depth, they are moved to the current depth. These values are considered for assignment to field prompts on subsequent .PROMPT primitives within the same command procedure. In batch mode, these leftover name/value pairs are written to the listing file as UNKNOWN.

4.5.3 Routines that Process SCI Primitives.

Each of the following routines processes an SCI primitive. It is assumed that you are familiar with the syntax of those commands. (Refer to the DNOS System Command Interpreter (SCI) Reference Manual for details.)

Processing of the .PROMPT primitive is described in the Command Procedure Processing Routines paragraph in this section.

4.5.3.1 XUSE.

XUSE processes the .USE primitive to redefine the five command libraries. The routine scans the access names and calls S$MAPS to do synonym substitution. The pathnames of the primary and secondary libraries are stored in a buffer pointed to by the variable USYS. In the absence an operand, a null pathname is stored in the buffer. XUSE sets a flag to force DMENU to search for a new menu the next time it is called. XUSE also sets the synonym $$CL to contain the specified list of pathnames.

The default command library is .S$CMDS, with no secondary libraries.

4.5.3.2 XPROC.

XPROC processes the .PROC primitive to install a command procedure into the primary procedure library. The name of the primary library is constructed by appending the name supplied with the primitive to the directory name stored in USYS.

XPROC calls XDATA (at entry point PUTDAT) to copy records between the .PROC and .EOP primitives into the file with the following pathname:

```
<USYS>.name
```

where:

@USYS	points to the name of the directory for the primary command library.
name	is supplied by the user on the .PROC primitive.

Comments (lines with * in column 1) are not copied. Leading blanks and text beyond the SCI delimiter are deleted.

4.5.3.3 XEOP.

XEOP processs the end of command procedure condition. The procedure depth counter is decremented and LIBSCN is called to switch control back to the calling command procedure.

4.5.3.4 XMENU.

XMENU processes the .MENU primitive to identify the menu that is displayed in the major loop. XMENU sets the parameters that cause DMENU to display the specified menu. The variable S$$MNU is set to reflect the form of the command.

Command	S$$MNU	Display Action
.MENU	-1	Does not display menu
.MENU name	+1	Forces display of menu
.MENU *name	0	Displays menu only in VDT

In the latter two cases, name is used to build the pathname of the file from which the menu is obtained. Menus are expected to be stored in the same directory as the primary command procedure library. The constructed pathname is of the following format:

<USYS>.M$name

where:

USYS	points to the directory name.

4.5.3.5 XOPTIN.

XOPTIN processes the .OPTION primitive to alter three characteristics of an SCI session: PROMPT, MENU, and PRIMITIVE. When the .OPTION operand field includes a keyword in the assertive state (that is, no value assignment is made), the following default values are assigned:

Keyword	Feature	Default Value
PROMPT	Prompt displayed in the major loop	[]
MENU	Name of the menu displayed in the major loop (VDT only)	LC (List Commands)
PRIMITIVE	Accept primitives at depth level zero	YES

Only those features that appear on the command line are altered.

4.5.3.6 XBID.

XBID processes the .BID primitive. XBID processes keywords and sets up parameters for routine S$BID. Elements of the PARMS list are stored in the NCT. The name of each element is made up of three binary numbers -- two zeros followed by a number that is the position in the list occupied by the value. For example, the name of the first element on the PARMS list is 001. Its value is the first element on the PARMS list.

After the keywords are parsed, XBID executes the following steps:

1. Closes the TLF (if it is open).

2. If the primary input is a terminal, issues an SVC to close the terminal.

3. Calls S$BID

4. Opens the TLF and terminal, as required to restore prior status.

5. Deletes any synonyms created for PARMS specified with .BID.

6. If this is a call from QBID, sets error synonyms appropriately.

4.5.3.7 XQBID and XDBID.

XQBID process the .QBID primitive. XQBID is the interface between SCI and the routine S$QBID. XQBID processes the keyword list, verifies keyword values and calls the S$ routine to bid the specified task.

XQBID has an entry point for XDBID, the routine that processes the .DBID primitive. This entry point is used for entry to the Debugger. The specified task is bid and immediately suspended.

4.5.3.8 XRBID.

XRBID processes the .RBID primitive to bid a task, and subsequent .RBID primitives to restart that task.

XRBID calls routine S$RBID, which is part of SCI. (S$RBID is not in the S$SYSTEM shared procedure segment.)

S$RBID uses the synonym $$RBID to determine whether or not SCI is taking end action and to determine whether or not to delete the calling task from the table of active RBID tasks. The logic of S$RBID is described in the following metacode:

```
   IF SCI is taking end action
      THEN $$RBID=Y;
      ELSE $$RBID="";
   IF the task is not in RBIDAC
      THEN This is the initial bid.
           Issue an Execute Task SVC (>2B);
      ELSE The task is already in the job, in a suspended state.
           Issue an Activate Suspended Task SVC (>07);
           Issue an Unconditional Suspend SVC (>06) for SCI;

*****************************************************************
*  SCI is suspended.  The RBID task must call S$WAIT to        *
*  return control to SCI (via Activate Suspended Task SVC).    *
*****************************************************************

   IF $$RBID is non-null
      THEN Call ENTRY to make the entry in RBIDAC, if necessary;
      ELSE Call ENTRY to delete the entry in RBIDAC, if necessary;
   END;
```

Routines S$NEW and S$WAIT coordinate information essential to the .RBID function. S$WAIT is discussed in the section of this manual entitled Conventions and Libraries. S$NEW is documented in the DNOS Systems Programmer's Guide.

Routine XRSTAT provides reporting and termination services for SCI, with regard to RBID tasks. When it is called with a request for status, it returns the names of all .RBID tasks currently in the user's job. When it is called to terminate tasks, it sets $$RBID to a non-null value and allows each of the tasks in the RBIDAC list to perform end action.

The calling sequence for XRSTAT is documented in the code.

4.5.3.9 XDATA.

XDATA copies the records between the .DATA statement and the next .EOD statement to a file specified by the access name on the .DATA statement. If no access name is specified, the data is copied into the TLF.

The entry point PUTDAT is used by XPROC to copy command procedures. In this case, the delimiter is .EOP rather than .EOD.

XDATA returns an error if the end-of-file is encountered before the delimiter (.EOP or .EOD).

4.5.3.10 XEVAL.

XEVAL processes the numeric assignments of the .EVAL command. S$INT is called to convert the integer expression text to a binary number. S$IASC converts the binary number to decimal

ASCII digits. S$SETS is called to assign this value to the specified synonym name.

XEVAL processes name and value pairs until the SCI line delimiter is encountered.

4.5.3.11 XSHOW.

XSHOW processes the .SHOW primitive, causing the specified file or files to be displayed through a call or calls to S$SHOW.

4.5.3.12 XSPLIT.

XSPLIT processes the .SPLIT primitive. The keywords and values in the operand field are verified as character strings. S$SPLT is called to split the command stream into sets of keyword/value strings. S$SETS is called to store the name/value pairs in the synonym table.

4.5.3.13 XSYN.

XSYN processes the assignments of a .SYN primitive. It calls S$SETS to perform the binding.

4.5.3.14 XSVC.

XSVC processes the .SVC primitive. After parsing keywords and values, the specified SVC is issued. If an SVC error is returned, the return code processor is called using the Return Code Processor SVC. (Refer to the DNOS Systems Programmer's Guide for details of the return code processor SVC.) The condition code synonym ($$CC) is set.

At the beginning of XSVC is a table of SVC opcodes and I/O subopcodes that are not allowed. These restrictions are imposed in order to protect the integrity of SCI. (Refer to the DNOS System Command Interpreter (SCI) Reference Manual for a list of the opcodes and subopcodes that are disallowed.)

4.5.3.15 XIF.

XIF processes conditional statements. The routine evaluates clauses associated with the .IF, .WHILE, and .UNTIL primitives. XIF saves the information necessary to effect the .LOOP, .REPEAT, .ENDIF, and .ELSE primitives.

XIF identifies the beginning of the construct, evaluates the operands, and takes the appropriate action, depending on the result of the specified comparison.

Three positional parameters are parsed:

```
      <string1>  <relation>  <string2>
```

An attempt is made to evaluate each string as a numeric integer expression. If this is not possible, S$SCOM is called to compare them as strings.

Transfer of control is accomplished by placing the next command line to be processed into the command buffer. This involves one or the other of the following:

* Reading lines forward until the matching .ENDIF primitive is found

* Backspacing to the matching .LOOP primitive.

The IF/LOOP level is maintained by XIF. The line number of each .LOOP is stacked when a nested .LOOP is encountered. These line numbers are used to calculate the number of lines to backspace the procedure file when transferring control to the matching .LOOP primitive.

XIF calls GETRLN to parse the relation. The offset returned is used to access a mask in the table RLNVAL. If the result stored in the status register by S$SCOM is equal to the mask, the relation is true.

The data structures RLNTBL and RLNVAL are organized in such a way that the index to the inverse relation name and mask are computable. For example, the first relation in both tables is IS and the last relation is ISNOT; the second is EQ and the next to last is NE, and so on. This organization is exploited in the code that processes .WHILE and .UNTIL primitives. The same code that processes the .WHILE is used to process the .UNTIL, with the operator inverted (table offset complemented).

4.5.3.16 XELSE.

XELSE bypasses the .ELSE primitive clause. Command procedure records are skipped until an .ENDIF or .EOP is encountered. XELSE pairs .IF and .ENDIF primitives within the .ELSE clause and returns when an unmatched .ENDIF is encountered.

4.5.3.17 XENDIF.

XENDIF processes the .ENDIF command. The IF/LOOP level is decremented.

NOTE

The primitives discussed in the following paragraphs, .COPY, .STAGE, and .EOS, are currently disabled. (The processor address table entries in GETCMD are set to zero). The routines XCOPY, XSTAGE, AND XEOS are code that is never executed.

4.5.3.18 XSTAGE.

XSTAGE issues an SVC to the Name Manager, requesting the creation of a new stage. The stage depth counter is incremented.

4.5.3.19 XEOS.

XEOS issues an SVC to the Name Manager, requesting a return to the previous stage. The stage depth counter is decremented.

4.5.4 Parsing Routines.

Parsing routines are called in the major loop for command processing. Two routines prepare the command buffer for parsing and the remainder process the data. The routine that fetches SCI verbs (GETCMD) is similar to these parsing routines, but its function is to transfer control through a branch table. For that reason, GETCMD and GETOPC are described with the high-level routines that determine the major path through SCI. The parsing routines discussed here process character strings to arrive at a value.

4.5.4.1 Data Structures.

The major data structures used by the parsing routines are CBUFF, the command buffer, and CBPTR, a pointer to CBUFF. When processing field prompts, TYPTBL and a temporary buffer contain information about the prompts and their current values.

CBUFF is a 256-byte buffer where input is stored and intermediate processing results are sometimes kept. CBPTR points to the next character (in CBUFF) available for processing.

TYPTBL is a structure for storing abbreviated information about each of the 22 possible field prompts. Field prompts are treated as positional parameters. Information about the first prompt specified is the first entry in TYPTBL. Information stored in the structures KWTBL (pointers to field prompt names) and VALTBL (pointers to current values of field prompts) is also based on position of the field prompt.

TYPTBL is a 46-byte table. The first word contains an offset to the field prompt currently being processed. Information in the TYPBTL is stored in the address that is the sum of the table address plus the value of the first word in the table. The value of the first word is also used as an offset into KWTBL and VALTBL to access information about the field prompt currently being processed.

TYPTBL contains two bytes of information for each field prompt. The first byte consists of flags that are used to guide the processing of this field prompt. The flags indicate the following conditions:

* Whether or not the field prompt is required

* Whether or not the value can be a list

* Type alternation - whether the field prompt has more than one acceptable type (This bit is used only while building the list.)

The second byte contains a count of the number of data types that can be considered for the value of this field prompt.

All field prompt attribute information is stored in AUX2. AUX2 is a 348-byte structure that overlays three buffers that are not in use when AUX2 is in use. The first 256-byte block is dedicated to AUX2, and is followed by:

ACNM	- A 51-byte buffer used by XDATA
FSTNAM	- A 20-byte buffer used by XSPLIT
RSTNAM	- A 20-byte buffer used by XSPLIT

The total buffer size (348 bytes) is static, but storage within the buffer is dynamically allocated as field prompt type declarations are processed. The first 23 words of AUX2 are reserved for pointers. The remainder is allocated as required. Free space, if any, is at the bottom.

The structure of AUX2 is shown in Figure 4-2.

```
---------------------------------------------------------------
|    | Pointer to Free Space (PF)                             |
P    ----------------------------------------------------------
O    | Pointer to Field Prompt #1 information (P1)            |
I    ----------------------------------------------------------
T    ----------------------------------------------------------
E    .                       .                                .
R    .                       .                                .
S    .                       .                                .
|    | Pointer to Field Prompt #22 information                |
---------------------------------------------------------------
P1->| TYPE ID               | # bytes supplemental information|
     ----------------------------------------------------------
    | Supplemental information for Field Prompt #1 (Note 1,2) |
     ----------------------------------------------------------
    | TYPE ID               | # bytes supplemental information|
     ----------------------------------------------------------
    | Supplemental information for Field Prompt #1 (Note 1,2) |
     ----------------------------------------------------------
P2->| TYPE ID               | # bytes supplemental information|
     ----------------------------------------------------------
    | Supplemental information for Field Prompt #2 (Note 2)   |
     ----------------------------------------------------------
     .                       .                                .
     .                       .                                .
     .                       .                                .
     ----------------------------------------------------------
PF->|  FREE SPACE                                             |
     ----------------------------------------------------------
     .                       .                                .
     .                       .                                .
     .                       .                                .
     ----------------------------------------------------------
     ==========================================================
```

Note 1 - This structure is a case with two type declarations for field prompt #1.

Note 2 - Formats of the supplemental information fields are discussed with the parsing routines that build this table.

Figure 4-2 AUX2 Data Structure

4.5.4.2 Command Buffer Preparation.

GETLINE and TXTSUB set up the command buffer so that one or more of the parsing routines can be used to process input data.

GETLINE.

Routine GETLINE gets the next information to be placed into the command buffer. There are three entry points:

* GETLNE - Fetchs a 72-character command line and displays the character string that is the command line prompt (the default, or as specified with the .OPTION primitive).

* GETINP - Fetchs a 72-character command line and does not display the prompt.

* GETDAT - Fetchs an 80-character data line and displays the prompt.

Depending on the entry point, an end-of-line (EOL) byte is placed in either column 73 or column 81 of the buffer. The buffer is blank filled to the EOL. In the code, EOL is referenced as an exclamation mark (!), but it is not a printable character.

A record from the appropriate source, either a file or device (primary input) or a command procedure definition (secondary input), is read into the command buffer. If the session is in batch mode, commands from the primary input are written to the listing file.

TXTSUB.

TXTSUB makes two passes on the command buffer to remove extra blanks and to replace synonyms and field prompts with current values.

On the first pass, multiple blanks are reduced to single blanks, except when multiple blanks appear within a quoted string.

The second pass is a right to left parse. Substitutions are made as follows:

* For every occurrence of @, the routine S$MAPS is called to supply the value associated with the synonym whose name follows @. If the synonym is undefined, the synonym name is used as its value in the command buffer text stream.

* For every occurrence of &, the routine S$MAPK is called to supply the value of the command prompt that follows &. If the command prompt has no value, the null string is substituted into the command buffer text stream.

On each pass, TXTSUB processing is terminated when the EOL character is encountered.

4.5.4.3 Text-Handling Routines.

The remaining parsing routines are divided into those that build and access the AUX2 data structure to verify field prompt value assignments, those that parse in search of the value to assign to a field prompt of a known type, those that skip over delimiters and superfluous blanks, and utility routines used by all parsing routines.

Routines that build and access the AUX2 data structure to verify field prompt value assignments are of two types:

* Routines that parse to build data structures for field prompt value verification:

  - GETALT. Builds a data structure of alternative types for a given field prompt. This routine calls GETLST and GETRGI if the alternative types are themselves lists and/or ranges.

  - GETLST. Builds a supplemental data structure of list elements

  - GETRGI. Builds a supplemental data structure of range limits

* Routines that access AUX2 to verify the value(s) being considered for assignment to a field prompt name:

  - GETVER. Verifies data attributes for a field prompt value

  - GETELT. Verifies that the item being considered is an element of the specified list

  - GETRNG. Verifies that the value being considered is within the specified range.

Routines that parse in search of the value to assign to a field prompt whose type is known are as follows:

* GETNAM. Name type

* GETACN. Access name type

* GETSTR. String type

* GETINT. Integer type

* GETYNO. Yes/no type

Routines that skip over delimiters and superfluous blanks are as follows:

* GETCMA. Skips over a comma

* GETEOL. Skips blanks to the first nonblank character. Returns an indicator if that character is the EOL

* GETEQL. Skips over the equal sign

* GNB. Skips over all blanks to the next nonblank character.

Utility parsing routines are as follows:

* GETKEY. Keyword

* GETRLN. Relation

* GETSYN. Synonym name

* GETTYP. Type declaration

These parsing routines return through SDSERR if an error condition is encountered and return a condition code in register zero.

If the routine produces an error, the command buffer and pointer are restored to their values prior to the parsing routine call. The calling routine is responsible for taking the appropriate action, based on the error.

If, however, no error is encountered, CBPTR points to the first character of the next item to be parsed and register one points to the text of the value for the item just parsed. This value may or may not be stored in the command buffer. Previous characters in the command buffer are not guaranteed to be unaltered.

4.5.4.4 Table-Building Routines.

The routines described in the following paragraphs build the AUX2 table.

<u>GETALT</u>.

GETALT processes field prompt type declarations. The following information is stored in TYPTBL:

* Flags that indicate the following:

  - Whether or not alternate types are declared

  - Whether or not a list has been declared

  - Whether or not this field prompt is required to have a value

* The number of types declared for this field prompt

GETALT also makes entries in AUX2 that contain the following information as required:

* A one-byte identifier for the type

* Number of bytes of supplemental information. (This may be 0, as in the case of a YESNO type where there is no supplemental information.)

* Supplemental information:

  - Range

  - List

GETALT is called once for each field prompt, and processes all type declaration information. It calls GETLST and GETRGI if necessary. Overflow of the AUX2 buffer is detected and reported through SDSERR.

GETLST.

GETLST processes the elements of a set of acceptable values that a field prompt may have. Enclosure of the list in parentheses is checked and a syntax error is returned if a parenthesis is missing. The list is stored in AUX2 as supplemental information for the ELEMENT data type. The AUX2 format of the list is as follows:

```
<number of elements in list><byte count for first
element text><text of first element><byte count
for translation value><translation value of first
element><byte count for second element text><text
of second element>...
```

This information is used by GETELT to verify that the element specified is on the list.

GETRGI.

GETRGI checks for AUX2 table overflow and for enclosure of range information in parentheses. Error conditions are reported through SDSERR. GETRGI parses the command buffer and constructs a supplemental information data structure in AUX2 consisting of the 32-bit signed integer lower and upper limits.

This information is used by GETRNG to ensure that a value being considered for the field prompt is within the specified range.

4.5.4.5 Verification Routines.

These routines verify whether or not a specified value is acceptable, according to the information in AUX2.

These routines are structured as follows:

```
Save CBPTR;
Search CBUFF for specific data type.  (This includes
 syntax checking and verifying attributes of the value.)
IF no error
     THEN
          Set CBPTR to point to the character in CBUFF
           immediately following the delimiter
           for this type;
          Set register one to point to the parsed value;
     ELSE
          Restore CBPTR to its prior position;
          Set an error indicator;
Return;
```

<u>GETVER</u>.

GETVER verifies that the value to be assigned a field prompt has the proper attributes. It accesses the information stored in AUX2. GETVER processes a single value or a list of values, checking each value against all valid types.

Upon completion of GETVER, register one points to a translated value list in AUX3.

NOTE

Field prompts declared type STRING are not subjected to any tests by GETVER. No information is stored in AUX2, and any non-null value proposed for the field prompt is accepted as is.

<u>GETELT</u>.

GETELT parses an item in the command buffer that is expected to be an element of a list built by GETLST. The input item parsed is paired with the text of each element of the list. The approximate matching algorithm is used to determine whether or not they match. An error is generated if the item does not appear in the list or if it matches more than one element in the list.

When a unique match is found, the item (or the translation value, if one exists) is returned. Notice that the translation value may differ, perhaps drastically, from the information input by the user.

Upon return from GETVER, a register points to the value. If it is the user input, that value is in the command buffer. If it is a translation value, it is in AUX2.

GETRNG.

GETRNG parses the command buffer for what is expected to be an integer type and verifies that it is in the specified range. The bounds are inclusive so the value may equal either limit.

GETINT is called to parse the next item in the command buffer. S$INT is called to convert that ASCII representation to binary.

An error condition is set if the value is outside the range limits and control returns to the calling program through SDSERR.

GETNAM.

GETNAM parses for the name type. The only checking done is to ensure that the first character is alphabetic or $, and that the succeeding characters are alphanumeric or $. It should be noted that the term alphabetic includes the ASCII representations of the uppercase characters A through Z, plus [, \, and], as well as the Katakana character set. Alphanumeric includes alphabetic characters and the ASCII codes for the decimal digits 0 through 9.

The GETNAM routine uses a data structure that contains the limits of the internal representation of these character sets. It is called ALPHA. The structure consists of the ASCII representations of the limits of the following three ranges, each of which is continuous:

* A through Z, and [, \, and]
* Katakana
* 0 through 9

GETACN.

GETACN verifies the syntax of an access name. It calls PTHNAM to check the syntax of each node of the character string.

The delimiter for an access name type is any character that is not alphanumeric, as in the preceding paragraph.

<u>GETSTR</u>.

GETSTR parses the command buffer in search of a STRING type. Two types of strings are processed -- quoted and unquoted.

A quoted string begins with a quote ("), which is stripped. Subsequent adjacent quote pairs ("") are replaced with a single quote. An unpaired quote is the delimiter for a quoted string. It is also stripped.

Processing of an unquoted string includes reduction of multiple blanks to single blanks and deletion of leading and trailing blanks. The delimiters for unquoted strings are as follows:

* Exclamation mark - !
* Right parenthesis -)
* Equal sign - =
* Quote - "
* Comma - ,

The delimiter is stripped.

<u>GETINT</u>.

GETINT calls GETSTR to isolate the character string to be processed as an integer type. Any remaining blanks are removed. S$INT is called to evaluate the expression. This is done to detect errors, not to obtain the value. GETINT returns the character string that is known to be a legitimate integer type.

<u>GETYNO</u>.

GETYNO calls GETNAM to isolate the name to be evaluated as a YESNO type. Only the first character of the name is checked for Y or N.

Unless an error is encountered, register one points to the original byte count followed by either Y or N and the remainder of the original character string.

4.5.4.6 Cleanup Routines.

The following cleanup routines are used:

<u>GETCMA</u>.

This routine skips over an anticipated comma. If the comma is the next nonblank character, it is skipped and CBPTR points to the first character after the comma that is not a line delimiter or a blank. New lines are read if necessary. (That is, the comma is treated as a line continuation character by SCI.)

An error is generated if the next nonblank character is not a comma. In the error case, CBPTR points to the character that was expected to be a comma.

GETEOL.

This routine advances CBPTR to the next nonblank character. An error is returned if that character is not the SCI line delimiter.

GETEQL.

This routine parses the equal sign. If the first nonblank character is =, CBPTR is advanced to the next nonblank character after =. If the equal sign is not found, an error is returned and CBPTR points to the character that was expected to be the equal sign. New lines are read if necessary. (That is, the equal sign is treated as a line continuation character by SCI.)

GNB.

GNB advances CBPTR to the next nonblank character. No errors are returned. No new lines are read because the line delimiter appears just past the right margin in the command buffer.

4.5.4.7 Utility Routines.

The following utility routines are called by many parsing routines:

GETKEY.

GETKEY calls GETSTR to isolate the character string to be processed as a keyword or field prompt name. S$SCPY is called to store the string in an 82-byte dedicated buffer pointed to by the global variable KEYWRD.

GETRLN.

GETRLN supplies information about the relation to be considered. GETNAM is called to isolate the character string that represents a logical operator. RLNTBL is searched for the specified operator, using S$SCOM to compare strings. GETRLN returns (in register one) the displacement from the beginning of RLNTBL to the match. This displacement can be used to access information in RLNVAL, a table of masks for isolating the comparison result of interest for this operation.

<u>GETSYN</u>.

This routine isolates the character string that represents a synonym name. GETSYN recognizes two forms -- name and access name types.

GETSYN calls GETNAM first, and if no name is found, it calls GETACN and attempts to parse an access name.

<u>GETTYP</u>.

GETTYP is used by GETALT to parse the command buffer for legitimate type declarations. The anticipated type is scanned by GETNAM. A table containing the text for all types (TYPNAM) is searched for a match.

Register one is returned pointing to a location that contains the displacement to the match. (This displacement is also the displacement into the table TYPXFR to the address of the routine that processes the type.)

This routine parses only one type and must be called repeatedly in the case of alternate types.

4.5.5 Display Routines.

The following routines direct the writing of output to the appropriate device or file, depending on whether the SCI session is batch or interactive.

4.5.5.1 DLINE.

DLINE writes or displays one line of output to a file or device. The form of the line depends on the SCI mode.

* Batch - Generates a page eject and page header as required, using counters LINECT and PAGECT for lines and pages, respectively

* TTY - Writes a single record without page headers, followed by a carriage return and line feed

* VDT - Writes a single record on the bottom of the screen. No page headers, carriage return or line feed

DLINE requires an output buffer that begins on a word boundary, and is an even number of bytes in length, because the buffer is used in the SVC call block for the write. If the record length of the output file or device is greater than the buffer length, the record is truncated.

If the buffer is an odd number of bytes in length, the number of bytes output is rounded down. The first two bytes of the output buffer are always cleared by DLINE. If the output is to a device, and there are no control characters in the text, DLINE uses the first two bytes for carriage control.

4.5.5.2 DBATCH.

DBATCH is called when the SCI session is in batch mode. DBATCH writes the field prompt name/value pairs to the listing file.

DBATCH writes the names and values from KWTBL and VALTBL. The values in KWTBL and VALTBL are the same as those in the NCT because in batch, the user has no opportunity to make changes interactively.

4.5.6 Subsystem Support.

The following routines are interfaces to the MAILBOX and TINFO subsystems.

4.5.6.1 MAILBOX.

MAILBOX is a separate program that processes messages sent between tasks.

Interactive SCI uses the receive services of MAILBOX in the initialization and termination portions of the task, as well as in DERROR. MAILBOX is bypassed in a batch session. The interface routines that receive messages (MBRCV and MBRLS) are included in the SCI990 procedure segment. The interfaces to MAILBOX are independent of the specific implementation of the MAILBOX functions.

Pointers to two buffers must be passed to MAILBOX interface routines. One buffer is for the message and the other is for time and date. The first byte in each of the buffers must contain the length of the buffer.

MAILBOX interfaces set the first byte of each buffer to zero if no message is found.

Upon return, if register zero is zero, no errors occurred. If an error did occur, the following information is returned:

* Register zero is the error code.

* If the error code in register zero is >90FF, register two points to an SVC call block that contains the error.

<u>MB$RCV</u>.

MB$RCV is the user interface for accessing messages sent to the task through MAILBOX. It leaves all conditions set in such a way that more messages can be requested later.

In addition to the buffers (message, time and date), MB$RCV requires a token list. The token list is a list of MAILBOX addresses to be searched. Each token should correspond to one of the tokens specified when a message is sent. The token list has the following format:

```
<LIST LENGTH><TOKEN LENGTH><TOKEN>...
```

where:

LIST LENGTH	is the total number of bytes in the list that follows.
TOKEN LENGTH	is the length of the next token (maximum of 8).
TOKEN	is the character string.

A maximum of three tokens is specified. SCI passes two tokens -- the station ID and the user ID.

<u>MB$RLS</u>.

MB$RLS is the user interface that allows the caller to take any messages pending and terminate communication with the MAILBOX subsystem.

MB$RLS is called by SCI during processing of the .STOP primitive.

4.5.6.2 TINFO.

TINFO is a separate task that owns the system data structures that contain information about the status of terminals. It processes the following SCI commands: CM, KBT, MSG, MTS, SBS, SDT, and WAIT.

Only those routines that provide SCI read access to the system communication area (SCA) are included in the SCI990 procedure segment. This allows SCI to determine the mode of the terminal.

SCA$R reads terminal parameters. When this interface routine is called, register one points to a buffer that contains as its first four bytes STnn, where n is a decimal ASCII digit. Upon return, the buffer contains whatever information TINFO has concerning the station. The format of the station name is hard coded.

4.5.7 Utility Routines.

SCI uses two utility routines of its own, as well as several routines in the S$SYSTEM procedure segment.

4.5.7.1 HEXSYN.

HEXSYN converts a specified binary value to a five-character string. The first character, >, is followed by the ASCII representation of the four-digit hexadecimal number. This character string is then assigned as the value of the specified synonym.

Registers one and two are pointers to the synonym name and binary value, respectively. They are preserved.

4.5.7.2 APPROX.

APPROX applies the approximate matching algorithm to two strings. The rules of the algorithm are outlined in the DNOS System Command Interpreter (SCI) Reference Manual.

4.5.7.3 S$SYSTEM Routines.

SCI makes extensive use of the following routines in the S$SYSTEM procedure segment. They are divided into two groups -- the first group is routines documented in the DNOS Systems Programmer's Guide. The second group lists routines documented in the section of this manual entitled Conventions and Libraries.

The following routines are documented in the DNOS Systems Programmer's Guide:

S$IADD	Adds double precision
S$IASC	Converts binary to ASCII
S$INT	Converts ASCII to binary
S$MAPS	Maps the value of a synonym
S$NEW	Initializes a task data base
S$PFIL	Submits print request from user task
S$SCOM	Compares strings
S$SETS	Sets the value of a synonym
S$SNCT	Searches the NCT

S$TAD	Formats time and date
S$TERM	Terminates a task

The following routines are documented in the Conventions and Libraries Section of this manual:

S$FMT	Formats the interactive display
S$GKEY	Gets keyword value
S$KEY	Sets a name/value pair in the NCT.
S$MAPK	Maps keyword value
S$OPN	Is the same as S$OPEN
S$OPNX	Forces an open extend of the specified file
S$PKEY	Writes to interactive terminal and waits for a reply
S$RIT	Reads information from an interactive terminal
S$SKEY	Sets keyword value (special case)
S$WIT	Writes to interactive terminal
S$PCNT	Purges the NCT
S$SETK	Sets keyword value
S$WAIT	Suspends the calling task

4.6 INTERNATIONALIZATION

All character strings displayed to the user are declared in the module SCIPRC.

SECTION 5

TEXT EDITOR

5.1 OVERVIEW

The Text Editor is an SCI subsystem through which records in a file are modified, added, inserted or deleted.

The Text Editor task is written in assembly language.

The Text Editor is a co-resident task in a user job with an interactive SCI session. The Text Editor requires access to an interactive terminal. It is not supported in a batch environment.

This section describes the structure of the Text Editor, its files and data structures, and the processing of the edit commands and functions.

Refer to the DNOS Text Editor Reference Manual for a detailed description of the Text Editor user interface.

5.2 STRUCTURE

The Text Editor is composed of three segments:

* Procedure segment S$SYSTEM - Library of routines that is used not only by the Text Editor, but by other DNOS tasks, including SCI.

* Procedure segment EDITOR - Nonreplicatable segment that is procedural code and that performs text editing functions and editing commands as they are entered from the interactive terminal.

* Task segment EDITOR - Replicatable task segment of the Text Editor that contains the task transfer vector, volatile code, volatile information that contributes to the definition of the current state of the edit, and S$SYSTEM routine workspaces and DSEGs. Each time the Text Editor is invoked, a unique task segment is created in the user's job.

5.3 FLOW OF CONTROL

The major phases of a text editing session are outlined in the following paragraphs:

5.3.1 Invoking the Text Editor.

The Text Editor is invoked by SCI during processing of any command procedure that bids the program. Once an edit is active, control can switch back and forth between the Text Editor and SCI, via the RBID mechanism.

5.3.2 Initialization.

Initialization of the Text Editor begins with an escape clause. This traps any command procedures that bid the task outside an active edit. The escape allows those command procedures to be executed without actually editing a file. (Command procedures, as shipped by Texas Instruments Incorporated, do not RBID the Text Editor unless an edit is in progress.)

If the escape clause is not taken, initialization is performed as follows:

1. Initializes the Text Editor variables and data structures

2. Opens the input file (if one is specified)

3. Creates the work files

4. Displays the first page of the file to be edited (if one is specified)

5.3.3 Major Path.

Three kinds of processing are done during a text editing session:

* Device service routine (DSR) processing - Functions performed entirely by the DSR, and not requiring execution of Text Editor code

* Function processing - Functions that do not require parameters (for example, Up Arrow and ERASE INPUT)

* Command processing - Functions that require parameters (for example, Move Lines (ML) and Find String (FS))

Figure 5-1 and Figure 5-2 depict the processing path through the Text Editor. Note that edit functions are performed in E$EDIT in a loop that is only exited when the CMD key is pressed. When this happens, the Text Editor suspends and returns to SCI for command procedure processing. This suspension is accomplished by a call to S$WAIT, a routine in the S$SYSTEM segment. The Text Editor is reactivated by SCI.

```
TXTEDT:  Reset STAY flag                                    (Note 1)
         IF CODE is not 0                                   (Note 2)
            THEN IF CODE is less than 0
                  THEN CODE=-CODE
                       Set STAY flag
                 Call E$CMD$ to process the command;
            ELSE Set STAY;
         ENDIF;
         IF STAY flag is reset
            THEN Call E$WAIT to suspend;
         Call E$EDIT;


Note 1 - STAY is a register.

Note 2 - CODE is the CODE parameter on the .RBID statement.
```

Figure 5-1 Flow Through E$1ST

The treatment of the CODE value on the .RBID statement allows a command procedure to process an edit command and then either to reactivate an edit already in progress (CODE < 0) or return to SCI (CODE < 0).

```
E$EDIT:  IF initialization is required
            THEN Call INITIL;
EDT100:  Read next input;
            IF CMD
               THEN Call E$WAIT;
               ELSE Perform function;
            Update the display at the terminal;
         END Loop READ;
```

Figure 5-2 Flow Through E$EDIT

5.3.4 Termination.

Termination of the Text Editor involves disposing of the session information as specified by the user and terminating the task by a call to S$TERM. Unless the session is aborted, a new file is created using the input file (if one is specified) and the work files. This new file is renamed according to parameters passed in the PARMS list.

A major design goal of the termination processing was to make it virtually impossible for the user to lose data. A new file is created containing information from the input file and the work files. If the replace option is YES, the input file is deleted when the new file is renamed. Only then are the work files deleted. The only exposure to loss of data is during catalog manipulation of pathnames.

Should a system crash occur during any other phase of termination, recover edit processing can restore most of the information entered during the prior edit. Specific limitations of the recovery scheme are discussed in the paragraph on detailed design of recover edit processing.

5.4 COMMAND PROCEDURES

The Text Editor does not include the capability to prompt interactively; control is returned to SCI when the user must supply additional information for command processing. Command procedures in the SCI language are provided for this purpose. The command procedures collect data interactively and RBID the Text Editor with the appropriate parameter values on the PARMS list.

5.5 FILES

The Text Editor manages two work files, the MOD file and the TEXT file, in addition to the input file, which is optionally provided by the user.

The MOD file contains one entry for each line of the input file that has been displayed. Information in the MOD entry indicates what kind of change, if any, is made to the original input file record. The largest file that is processed by the Text Editor is one with 65,250 records. (The record number variable is one word.) The size of the MOD file is monitored for exceeding the maximum file size.

The TEXT file contains the text of the modified or inserted lines.

The MERGE file is created during termination of the Text Editor. After the revised file is built, it is renamed according to the parameters passed to the Text Editor with the termination request.

5.5.1 Input File.

The pathname of the input file is specified on the PARMS list passed to the Text Editor with CODE=0 to (re)activate an edit session. If a pathname is provided, it must be for a relative record file or a sequential file. Key indexed files cannot be text edited. An error is generated if an attempt is made to text edit a device or a key indexed file. Two other prompts related to input files are:

1. Exclusive Edit - If yes, the file is opened exclusive write. If no, the file is opened shared.

2. Length - Maximum length of lines in the file to be edited. Records longer than this are truncated. Records shorter than this are blank filled in the output file.

If an input file name is not provided, editing takes place with regard to the MOD and TEXT files only. The presence or absence of an input file causes no significant changes in processing.

5.5.2 TEXT File.

The TEXT file is used to store the text of inserted, changed or moved records. It is a relative record file with a logical record length as prompted for in the XE command, not to exceed 240 characters. The TEXT file is created during initialization of the edit. Its pathname is as follows:

.S$TEXTxx

where:

xx is the station ID at which the edit is active.

The structure of the TEXT file is shown in Figure 5-3. TEXT file records are written sequentially and are not blocked in physical records.

FIRST RECORD:

Byte(s)	Contents
0	Pointer to end of pathname. Fixed at >34 to reserve 52 bytes
1	Number of bytes in input file pathname
2-53	Pathname of input file
79-80	Time stamp

ALL REMAINING RECORDS:

Byte(s)	Contents
1-LENGTH	Text of a line - where LENGTH was prompted for an XE command.

Figure 5-3 TEXT File Format

The two-byte time stamp is the result of an exclusive or of the seconds, minutes, hours, date (Julian), and year returned from the time and date SVC.

The time stamp is created during the initialization of an edit session. In addition to being written in the first record of the TEXT file, it is written in each physical record of the MOD file. The time stamp is used as a validity check when merging the files to create the output file. It is used by Recover Edit to verify the mod and text records are not information left over from another edit session.

5.5.3 MOD File.

The MOD file is used to record the types of changes made to the input file and to point to data in the TEXT file.

If an input file is specified, there is one MOD entry for each record of the input file. As each record is displayed for the first time, a null MOD file entry is written for the displayed (or skipped over) input file record. Additions and insertions are recorded by writing MOD file entries with numbers greater than the number of lines in the input file, and linking them with the MOD entries that correspond to the appropriate input file records. As part of initialization of a session, the characteristics of the input file are read, and the number of records originally in the input file is stored in the variable LSTINP. MOD file entries for inserted lines begin at LSTINP+2. Entry number LSTINP+1 is used for the end-of-file (EOF) line. In the current implementation, the variable MODEOF is equal to LSTINP+1.

If an input file is not specified, MOD entries are assigned sequentially, and the MOD file becomes a complete linked list of entries.

The MOD file is a relative record file with a logical record length of 252. The value 252 is chosen to best utilize space on disks with a 256-byte sector size. Twenty-five entries are blocked into a logical record. The remaining two bytes contain the same time stamp that appears in the first record of the TEXT file. The structure of each MOD file entry is shown in Figure 5-4.

The MOD file is created during initialization of the edit. Its pathname is as follows:

```
.S$MODxx
```

where:

xx is the station ID at which the edit is active.

```
BYTE
         -----------------------------------------------------------
  0     |              Previous MOD file entry number               |
        |-------------------------|---------------------------------|
  2     |              Next MOD file entry number                   |
        |-------------------------|---------------------------------|
  4     |              This MOD file entry number                   |
        |-------------------------|---------------------------------|
  6     |              TEXT file record number                      |
        |-------------------------|---------------------------------|
  8     | MOD type code           |    Link flags                   |
         -----------------------------------------------------------
```

BYTE(S)

0-1	Previous entry number. If this MOD entry is linked to a previous entry, the number is stored here.
2-3	Next entry number. If this MOD entry is linked to a next entry, the number of that entry is stored here.
4-5	File entry number of this MOD entry. This is provided as an error check.
6-7	TEXT File record number. If this is an insert or a change record, the text of the change is stored in the TEXT file record specified by this number.
8	MOD type code. This value determines whether the input file record corresponding to this MOD entry, or the inserted or changed text is to be deleted. An unmodified input record is denoted by a null code.

0 - Null	2 - Delete
1 - Insert	3 - Change

9 Link flags. In most cases, if a MOD entry is linked, it is linked both forward and backward. However, if a line is inserted at the beginning of the file or just prior to the end-of-file, the corresponding MOD file entry is only linked in one direction.

>80	Linked to previous entry only
>40	Linked to next entry only
>C0	Linked in both directions

Figure 5-4 MOD File Entry

5.5.4 MERGE File.

The MERGE file is created during termination of the edit. It is a temporary file so its pathname is autogenerated by DNOS.

The characteristics of the MERGE file are determined by the following algorithm:

```
IF Specified output file already exists
   THEN Use the output file characteristics;
   ELSE IF an input file is specified
         THEN Use input file characteristics;
         ELSE Use default sequential file characteristics;
ENDIF;
```

If the logical record length of the file whose characteristics are being duplicated is greater than 80, records in the input file are blank filled past column 80.

The MERGE file is created on the same disk volume as the output file so that the rename SVC can be used. The following logic is used to determine the name of the volume on which the file is created:

```
IF the pathname starts with a .
    THEN Build MERGE file on system disk;
    ELSE
       Issue map logical name SVC
       IF Value returned is null
          THEN Volume name is first node of pathname;
          ELSE Volume name is first node of logical name value;
ENDIF;
```

5.6 DATA STRUCTURES, VARIABLES AND SYNONYMS

Resident data modules are linked into the task segment. The modules E$DDTA and E$FDTA contain data used strictly by the Text Editor.

5.6.1 Data Related to the Display.

The module E$DDTA contains, in general, data pertaining to the state of the display, buffers used by some of the commands, and 80 bytes of patch space, as follows:

* EDTFLG - Flag indicating whether or not an edit is in progress. A nonzero value indicates an active edit.

* TXTBUF - General purpose buffer for reads and writes

* LINBUF - Buffer used for building lines to be displayed to the interactive terminal. This buffer overlays TXTBUF. The format of LINBUF is as follows:

Byte(s)	Contents
0-4	Line number. Five ASCII characters Blank if this line is an insert. Two bytes preceding buffer are reserved for carriage control.
5	Field size. Must be set after each read operation
6-241	Text (242 characters). Byte 6 must be on a word boundary. Two bytes at end are reserved for carriage control.

* BACKUP - Used to save the line for back out processing

* BUFFER - 250-byte buffer for blocking 25 MOD file entries

* CLBUFR - Line compare buffer containing the text of the current cursor line as it was when the cursor was first moved to that line, or when the last TEXT file record was written. The information in the compare buffer is used to determine whether a change record is processed in the MOD/TEXT files for this input record.

* MLBUFR - Working copy of current line containing all changes made to date

* PATCHE - 80 bytes of patch space available for data that must be patched into the resident module until a source change is made

5.6.2 Data Related to Text Editor Files.

Module E$FDTA contains the following data, used, in general, the file management modules of the Text Editor:

* I/O Request Blocks (IRBs) for the input, MOD, TEXT, an MERGE files

* File condition flags that indicate whether or not th MOD, TEXT and input files are opened

* INPFIL - Flag that indicates whether or not an input file is specified by the user

* CURPOS - One-word data structure that contains the cursor row in the leftmost byte and the cursor column in the rightmost byte. Always relative to LHSCOL, not to screen edge.

* LHSCOL - Lefthand side column (0-origin)

* LINTBL - Line table with one entry for each displayable line, plus entries for a header and a trailer line. The header line is a dummy record that is sometimes used as a temporary storage area. The trailer line is used to store text to be displayed as the EOF line.

  Each entry in LINTBL is a copy of the MOD file entry corresponding to the edited line displayed on that line of the terminal. See Figure 5-4 for the format of each entry. The entries in LINTBL are structured exactly like entries in the MOD file. The Text Editor file management and I/O package maintains this table. Information regarding changes that have been made to the text of a line is recorded in this table. The table contains entries for displayed lines only, and is not equivalent to the output buffer for the MOD file.

  The size of LINTBL is determined at run time. A read device characteristics operation for SVC >00 is issued to determine the number of displayable lines available on the terminal. A Get Memory SVC is issued to obtain adequate space for LINTBL. The number of lines on the interactive terminal is kept in the variable VDTSIZ.

Note that the full text displayed on a screen is not kept in memory (in the Text Editor address space), but is constructed on a line-by-line basis, as required, from input file, MOD file, and TEXT file records.

5.6.3 Synonyms.

Only one synonym, $$EA, is specifically accessed by the Text Editor. $$EA is used to indicate whether an edit session is currently active. Just prior to returning control to SCI, the Text Editor sets this synonym to Y if the session is not being ended. The Text Editor then calls S$WAIT to suspend until the task is RBID. When processing a request that terminates the session, $$EA is set to N just prior to the call to S$TERM. The name of the synonym ($$EA) is hard coded.

5.7 FILE MANAGEMENT AND FILE I/O

The Text Editor file I/O package is the vehicle by which all but exceptional I/O functions are implemented. This simplified I/O interface leaves the edit function and command processors free to consider I/O on the basis of lines. It frees the processors from consideration of the details of I/O in each specific circumstance.

File management routines in module E$FMNG determine which record in which file represents the desired line, and direct the I/O routines in module E$FLIO to perform the details of the function specified.

NOTE

> The file management package discussed here serves a different purpose than the functions implemented by the operating system File Manager. Despite the similarity of names, the two are separate entities.

The E$FMNG interface is used whenever the file being edited (that is, the logical merging of the input, MOD and TEXT files) is affected. I/O routines in the module E$FLIO are called directly when action is required on any other file. These routines have a BLWP/RTWP interface.

The control entry point for the package is E$FMNG. The function code passed in register 1 determines which process is performed.

Function Code	Process Performed
5	Position to beginning-of-file
6	Position to end-of-file
7	Page forward (up) n records
8	Page back (down) n records
9	Read record for line n
10	Change record for line n
11	Delete record for line n
12	Insert record for line n

Line numbers, pointers to pathnames of files, and pointers to buffers are passed in registers, as noted in the subsequent paragraphs. Specific register assignments are documented in the code. Error indications are returned in register 0. (A value of 0 indicates no errors.) If I/O to a file produces an error, E$WAIT is called to return directly to SCI. SCI displays the message.

All edit and command functions conduct I/O on the basis of displayed lines. I/O operations include read, insert after and change a specified line, among others. The line number is derived from the cursor position for VDT terminals and is always 1 for TTY terminals.

All file management functions use the data structure LINTBL (the line table, discussed in the paragraph on data related to Text Editor files).

When E$FMNG is called to read a line, the edited line is placed into the specified buffer. The edited line is found by using the MOD file entry, which points to a line in the input file, or points to an inserted line in the TEXT file, or indicates that the line has been deleted. The text of the line is accompanied by the input file record number (blank if the line is an insertion). Therefore, the read operation makes all data to be displayed available to the calling routine.

The discussion of each function includes the parameters passed with the request. Specific register assignments are documented in the code.

5.7.1 Change Record for a Line.

Input to this process is the line number (in the edited file) on which the change is detected, and a pointer to the buffer of new text of the line.

If the record has been previously changed, the existing TEXT file record (whose file record number is in the LINTBL entry) is rewritten with this new change.

If the MOD code for this line is null (indicating no corresponding TEXT file record exists), the next available TEXT file record number is obtained, and the new text is stored in that TEXT file record. The MOD code in the line table is altered to reflect that this line has been changed.

5.7.2 Delete Record for a Line.

This function, implemented in E$FDEL, marks the line corresponding to the current line in the line table as a deleted record. The appropriate entry is updated in the MOD file. The specified entry in the line table is deleted by writing the entry for the following line over the deleted line, then moving all subsequent entries in the line table forward one line. The next record in the input file is inserted at the bottom of the line table unless the corresponding MOD file entry specifies that the line has been deleted. Deleted lines are skipped until the next nondeleted line is found.

5.7.3 Insert Line.

This request must include the line number and a pointer to the buffer of text to be inserted. The insert line request is not processed if the number of records in the MOD file is greater than 65,250. If the MOD file is not full, it is determined whether or not the current MOD entry is chained back to the previous entry. If not, they are chained. If so, the previous MOD entry is read and the MOD entry for the line to be inserted is built and linked between the previous and the current entries. All three entries, now linked, are rewritten to the MOD file. All line table entries from the current entry through the end of the line table are moved down in LINTBL. The preceding entry is read and written over the current entry. This effectively inserts a MOD file entry and its corresponding TEXT file record before the old current line.

5.7.4 Open Files.

The routine E$OPEN is called directly by the Text Editor rather than through E$FMNG. A pointer to the input file pathname is passed to the routine. After a check to ensure that the pathname does not reference a device, the input file, if any, is opened. If the input file characteristics are acceptable, (that is, if it is not a key indexed file) the TEXT and MOD files are created by E$CRET. Flags are set to indicate that the files are opened, and whether or not an input file is specified. Control returns to the calling program.

5.7.5 Page Back.

This routine calls FGTRPV to get the previous record for line zero, and repeats the call for the specified count. This routine does not deal with line numbers as page forward does.

5.7.6 Page Forward.

Page forward is called with the relative roll count and the absolute line number. This routine operates in two modes -- page forward a certain number of records, or continue to page forward until a certain absolute record number is found. In the latter case, if that record is not found, an error code is returned. (If an error message is to be generated, the calling routine must do it.) The subroutine FGTNXT moves line table entries up one and reads the MOD entry corresponding to the next line of the edited file into the bottom of the line table.

Table 5-1 E$FLIO Routines Summary

Entry Point	Process	Parameters
E$OPEN	Moves access name into IRB (input file only) Opens input file	Address of IRB Address of access name
E$CRET	Creates specified file	Address of IRB
E$CLOS	Closes specified file Releases LUNO	Address of IRB
E$DELT	Closes specified file Releases LUNO Deletes file	Address of IRB
E$REWD	Rewinds specified file	Address of IRB
E$READ	Reads a record from specified file	Address of IRB File record number Address of buffer
E$WRIT (Note 1)	Writes from buffer to specified record in the specified file (Note 1)	Address of IRB (Note 1) Record number Address of buffer
E$GTFC (Note 2)	Reads characteristics of file Validates file type Checks write protection Returns file type (Note 2)	Address of IRB
E$WEOF	Writes EOF in the specified file	Address of IRB

Note 1 - E$WRIT does not write in the input file. The IRB must specify the MOD, the TEXT or the MERGE file.

Note 2 - E$GTFC returns a flag in register two of the caller's workspace. The values indicate file type, as follows:

1:Relative record

2:Sequential

5.7.7 Position at Beginning-of-File.

Position at beginning-of-file has no parameters. E$REWD is called to rewind the input file. The line table is cleared and the first MOD file entry is read into the line table entry. All entries are moved up one slot until the first entry has been moved into line zero of the line table.

5.7.8 Position at End-of-File.

Position at end-of-file has no parameters. The entry point is E$FEND. The routine calls E$GMOD to read MOD file entries until the MOD file EOF is in position zero of the line table. FGTPRV is then called to position the EOF entry into line one and the previous line in line zero.

LCOUNT is set to the number of active (not deleted) records in the edited file.

5.7.9 Read Record for a Line.

Routine E$FRED reads the text for the specified line into th buffer. If the MOD code in the corresponding entry is null or change, the record number is converted to ASCII and returned with the text. If not, the record number field is blank. If the MOD file entry in the line table for the specified line is null, the text is read from the input file. If it is a change or insert, the text comes from the TEXT record specified by the line table entry. An EOF indication is returned if the specified line number is not found.

E$FRED is called with the line number and a pointer to an 88-character buffer, which must begin on a word boundary.

5.7.10 E$FLIO Routines.

Both the Text Editor and E$FMNG call E$FLIO routines for specific I/O requests. Table 5-1 summarizes E$FLIO entry points, functions and parameters.

Except where noted in Table 5-1, E$FLIO routines use a workspace that is separate from the caller's workspace.

5.8 DETAILED DESIGN

Linkage between edit subroutines is implemented with a return address stack. The stack and the code that manages that stack are in the module E$STAK. The calling routine is STCALL and the return routine is STRETN. No registers are saved or restored. As many as ten calls can be stacked at any time. If the call being processed overflows the stack, STCALL drops into an infinite loop. If the return being processed underflows the stack, STRETN drops into an infinite loop.

Edit functions are not concerned with file record numbers. Command functions are concerned only to the extent that the lines of the edited file (the logical merging of the input and TEXT files as specified by the MOD file) must be positioned appropriately for display during the course of command processing.

5.8.1 RBID Statement Parameters and CODE.

The RBID statement that invokes the Text Editor is unique for each service requested. All calls go through the entry point TXTEDT in module E$1ST. The CODE value and PARMS passed on the RBID statement provide the information the Text Editor needs in order to perform the requested operation. Table 5-2 shows the values of CODE for the various requests.

Table 5-2 CODE Values for Edit Requests

CODE	Request
0	(Re)activate Text Editor (XE, XES)
	Modify Tabs (MT)
	Modify Roll (MR)
	Modify Horizontal Roll (MHR)
	Modify Right Margin (MRM)
1	Show Line (SL)
2	Copy Line(s) (CL)
3	Move Line(s) (ML)
4	Delete Line(s) (DL)
5	Find String (FS)
6	Replace String (RS)
7	Delete String (DS)
8	Insert File (IF)
9	Quit Edit (QE)
10	Save Line(s) (SVL)
11	Recover Edit (RE)

5.8.2 E$WAIT.

This routine, in the module with the same name, calls the appropriate S$ routine to suspend or terminate the Text Editor.

E$WAIT does the following processing:

1. Calls S$XFER if there is an error condition to report.

2. Closes the LUNO to the interactive terminal.

3. Calls S$XFER if registers must be set before calling S$WAIT or S$TERM.

4. If an edit is in progress (as indicated by the flag EDTFLG), the Text Editor is suspended by calling S$WAIT. Otherwise, the task is terminated by calling S$TERM.

After calling S$WAIT, when the Text Editor is RBID, E$WAIT opens (with event characters) the LUNO to the terminal and branches to TXTEDT in E$1ST.

5.8.3 E$DISP.

The routine E$DISP writes to the display for either VDT or TTY terminals. The processing includes adjusting the cursor position when line numbers are displayed. E$DISP is called with two arguments. One indicates how much of the display is to be rewritten, as follows:

Argument	Action
<0	Refreshes whole display
=0	Refreshes from current line to bottom of display
>0	Refreshes from top of display to current line.

The other argument indicates whether the current line is to be read or whether the text of the line in MLBUFR is to be used.

DISPAG contains a loop in which the line is read (by call to the file management and I/O package) and rewritten. The loop is exited when all lines have been rewritten or when an EOF is encountered. When the EOF occurs, the current line is saved as the EOF line, and the remaining lines on the display are cleared.

Within the loop to read lines, when the line that matches the row portion of CURPOS is read, SAVLIN is called to save the text in CLBUFR for later compares.

5.8.4 Edit Functions.

Edit functions pertain to the insertion, deletion, and modification of data on the cursor line as it is displayed at the terminal. Edit functions are invoked by keyboard keys, and have no parameters except as implied by the cursor position at the time the function is invoked.

Edit functions are described only to the point that I/O services are required. Interfaces with the Text Editor file management and file I/O package are noted. Edit functions not discussed in this section are performed by SVC. For further details on I/O to a specific device, refer to the DNOS Supervisor Call (SVC) Reference Manual.

Edit functions vary, depending on the Text Editor state and mode. As used in this discussion and in the code itself, the Text Editor operates in two states -- VDT and TTY. The VDT state exploits the advantages of the 911 VDT and the 911-like behavior of other VDTs. The TTY state is for the 820 terminal and the 733 ASR terminal. The variable STATE is accessed (through S$ routines) to determine the state of the terminal.

The editing mode governs the action taken with selection of the new line function, which is requested by pressing the RETURN key. In edit mode the cursor is positioned on the first tab position of the next line, with a roll-up of one line taking place, if necessary. In compose mode, selection of the new line function causes the introduction of a blank line following the current line, with the cursor positioned as previously described.

Edit function processing begins in the module E$EDIT at the label EDT100. General flow through E$EDIT is shown in Figure 5-2. This code sets up the parameters for a BLWP to S$RIT, which does all reading from the interactive terminal. Factors considered during this setup include the state of the terminal, tab information, right margin position, and whether or not the cursor is on the EOF line of the display.

E$EDIT checks for an event character that physically moves the cursor to the next line. This does not signify completion of processing on the current line i.e., the line for which the read was issued. This situation is detected when the DSR returns a line number that is not the line number for which the read was issued. Corrective action is taken -- the cursor is set to the last displayed column of the line number for which the read was originally issued.

After all special conditions have been processed, the event character returned by S$RIT is used to look up the address of the subroutine that performs the function indicated. The address of the required routine is written into a branch instruction in the task segment that makes the subroutine call.

Following completion of processing by the edit function processor, control is returned to E$EDIT. A branch back to EDT100 is made and the Text Editor prepares for the next edit function. An exception to this return occurs only if the CMD function is selected. In this case, the EDT100 loop is exited by calling E$WAIT to suspend the Text Editor and reactivate SCI.

The edit functions can be grouped into the following categories:

* CMD - Exit the edit function loop

* Toggle switches - Switch line number display status or switch edit mode or switch word wrap mode

* Cursor and roll functions - All movements of the cursor except cursor right and cursor left, which are sometimes processed by the DSR. Cursor left is processed by the DSR only if the lefthand edge of the screen is column 1. Cursor right is processed by the DSR only if the right margin is on the screen.

* Tabbing operations

* Line functions

5.8.4.1 CMD.

This function is processed in E$EDIT. When it is selected, the Text Editor is in a state to return control immediately to SCI990. E$EDIT calls E$WAIT which calls S$WAIT to suspend the Text Editor.

5.8.4.2 Edit/Compose.

The edit/compose logic switches the editing mode between compose mode and edit mode. The necessary cleanup operations are also performed. Processing begins at the label MODFLP in the module E$KCL1. The edit/compose flag is inverted. The word wrap flag is reset if you are changing to edit mode.

5.8.4.3 Line Number Display.

The line number display function inverts the state of DLNFLG, the flag that indicates whether or not the line numbers are displayed at the left of each line. When the Text Editor is initialized, DLNFLG is set to 1, which indicates that line numbers are displayed. Any other value of DLNFLG causes suppression of line numbers.

Processing begins at NUMFLP in E$KCL1. The current cursor column is adjusted to reflect the requested change if necessary. The cursor is adjusted if you are changing from no line number display to line number display and the cursor is on a character that would be pushed off screen. The flag is then reversed, E$DISP displays the current page, and if the state is TTY the portion of the current line up to the cursor is rewritten. Control is then returned to E$EDIT.

5.8.4.4 Cursor Down.

The cursor down function operates virtually the same in both states.

The entry point is CURDWN in E$KCL1. Upon entry, a test is made to determine if the cursor is currently on the EOF line. If so, the operation is ignored. The current line is checked and a mod record written if needed via COMSAV. If the cursor is not on the EOF line, but is on the bottom line of the display, (which is always the case in TTY mode), the display is rolled up, or paged forward, one line. E$DISP is called to rewrite the display and save the new line for later compares.

If the cursor is not on the bottom line of the display (VDT only), the cursor row address is incremented by one. E$FMNG is called to read the next line of the edited file. SAVLIN is called to save the text for later compares. The display is not rewritten. (The cursor is moved down by the next call to S$RIT.)

Control is returned to E$EDIT.

5.8.4.5 Cursor Up.

The current line is checked and a modification record is written, if needed, using COMSAV.

The cursor up entry point is CURSUP in E$KCL1. Upon entry, a test is made to determine the state of the terminal.

In TTY mode, E$FMNG is called to page back or roll down one line (a NOP if the first record is displayed or if the file is empty) and the display is rewritten.

In VDT mode, if the cursor is on the top line, the display is rolled down, or paged back one line. E$DISP is called to rewrite the display and to save the top line for later compares. If the cursor is not on the top line, the cursor row address is decremented by one. E$FMNG is called to read the previous line and it is saved for later compares. The display is not rewritten. (The cursor is moved to the previous line by the next call to S$RIT).

Control is returned to E$EDIT.

5.8.4.6 Home Cursor.

The record corresponding to the top line of the display is read and saved for later compares, if the cursor is not already on the top line (always true in TTY mode), and the cursor position is set to the first tab position for the next S$RIT call to be made by E$EDIT.

5.8.4.7 Roll Down.

Roll down processing begins at ROLDWN in E$KCL1. E$FMNG is called to page back the number of lines currently in the roll parameter. E$DISP displays the rolled page. On a TTY terminal, the top line of the rolled page is displayed.

5.8.4.8 Roll Up.

Roll up processing begins at ROLLUP in E$KCL1. E$FMNG is called to page forward the number of lines currently in the roll parameter. E$DISP displays the rolled page. On a TTY terminal, the top line of the rolled page is displayed.

5.8.4.9 Tabbing Operations.

The following routines in E$TABS are used in processing tab operations:

* NEXTAB - Always permits TAB wrap around. Entry point GETNXT does not permit TAB wrap around and positions the cursor to 1 character past the margin if no tab is found to the right of the current position. NEXTAB updates the cursor position to the column in which the next tab stop occurs.

* PRVTAB - Examines the cursor position and returns the column in which the previous tab stop occurs. If there is no tab stop between the current cursor position and the beginning of the line, cursor position is set to the last tab stop prior to the right margin.

VDT State Tab Operations.

Clear to tab processing begins at CLRTAB. A line of blanks is set up, a call to DUPTAB is made to write the appropriate number of blanks from the current cursor position to the next tab position.

Duplicate to tab processing begins at the label DUPTAB. If the cursor is on the EOF line, the operation is ignored. If not, E$FMNG is called to read the preceding line in the file. GETNXT is called to obtain the next tab position, with tab around not permitted. The start position of the cursor is used to determine the buffer starting address for the write and a call to S$WIT is made to write the change. Control is returned to E$EDIT.

Tab back processing begins at the label TABACK. A call to PRVTAB sets up the cursor position, based on the tab position to the left of the current cursor position.

Tab forward processing begins at the label TABFWD. NEXTAB is called to set the cursor position on the tab position to the right of the current cursor position.

5.8.4.10 Clear to End-of-Line.

The entry point for the clear to end-of-line function is KPSKIP in the module E$KCL2.

The original line is saved for possible later backout. The buffer to the right of the cursor is blanked and the line is rewritten.

5.8.4.11 Delete Line.

The entry point for the delete line function is DELLIN in the module E$KCL2. The line specified on the last S$RIT call is deleted. A call to E$FMNG deletes the line, the display is rewritten, and control returns to E$EDIT.

5.8.4.12 Insert Line.

Processing for insert line begins at INSLIN in the module E$KCL2. A blank line is built in the compare buffer and E$FMNG is called to insert that line before the current line. Finally, E$DISP is called to display the new page with the inserted blank line.

5.8.4.13 RETURN.

The new line function operates differently for different combinations of mode and state. Processing begins at RETURN in E$KCL2, where terminal state is determined.

<u>VDT</u>.

E$EDIT fixes the cursor correctly. COMSAV in E$CSAV is called to determine if there have been any changes in the current line. If so, a change record is written into the MOD file and the text is written to the TEXT file. Mode is checked.

<u>Edit</u>.

The cursor column is set to the leftmost tab, and CURDWN in E$KCL1 is called to complete new line processing. CURDWN is described in the Cursor Down paragraph.

Compose.

Special processing takes place if the EOF is displayed on the current page. E$FMNG is called to roll the display up one line, a blank line is inserted at the current cursor position. Control is then returned to E$EDIT.

5.8.5 Command Functions.

With the exception of the request to initiate an edit session, command functions are processed starting in E$CMD$. Parameter values must be passed to the Text Editor in the PARMS list of the .RBID statement.

The selected command is specified to the Text Editor by the CODE parameter as detailed in Table 5-2.

E$CMD$ verifies that the CODE value passed is a recognized value, enforces rules concerning whether certain requests are valid with the current state of the Text Editor, and reports errors due to unacceptable values in the PARMS list.

When E$CMD$ is called, CODE has a value greater than zero.

E$CMD$ checks the internal flag EDTFLG to determine whether an edit is currently in progress. If an edit is in progress, control is transferred to the specific command processor through a branch table. If an edit is not in progress, only CODE values of 0 (initiate a session) or 11 (recover a session) are processed. This condition is enforced in the code. All other requests cause a return to SCI with an error condition set. A recover edit request is aborted if a session is currently active.

Each command processor accesses the PARMS list as needed and deals with cases in which the values are not appropriate for the particular edit in progress. Invalid parameters may cause control to be returned to E$CMD$ for the display of an appropriate error message. In all other cases, each command processor is responsible for the termination of its processing with or without the display of an error code and/or descriptive message. Command processors return to E$1ST by a RTWP instruction unless the error return to E$CMD$ is taken.

Command requests can be grouped into three categories of similar function:

* Session commands - Those that initiate, terminate and recover an edit session

* Independent commands - Those that cause no significant activity within the Text Editor

* Line and string commands - Those that alter one or more lines in the file

NOTE

The command functions involve parameters that are field prompts in SCI command procedures. The name of each parameter on the PARMS list is the same as the name of the respective field prompt. Refer to the <u>DNOS Text Editor Reference Manual</u> for a detailed discussion of the parameters.

5.8.5.1 Session Commands.

These commands activate, reactivate, terminate or recover an edit session.

<u>Activate Session</u>.

SCI Commands: XE, XES

CODE: 0

PARMS:

Number	Definition
1	File access name
2	Number of lines to roll
3	Right margin position
4	Scaling? A value of 1 suppresses the scale display
5 through n	Tab columns

The command procedure that issues the request to initiate an edit session is responsible for ensuring that no edit is in progress on a file other than the one specified. Otherwise, the Text Editor resumes the previous edit, disregarding the input file access name in the PARMS list.

Initiate session (the Execute Editor (XE) or Execute Editor with Scaling (XES) command) processing starts in the module E$EDIT. EDTFLG is checked to determine if an edit is currently in progress. If so, initialization is skipped.

If an edit is not in progress, a check for batch mode is made. The Text Editor terminates if the bidding task is in a background job. The error is reported through S$XFER.

The module E$INIT is called to initialize variables in the resident modules E$DDTA and E$FDTA, respectively.

E$INIT initializes the session in the edit mode, unless no input file is specified (in which case, compose mode is chosen). The session is initiated with line numbers displayed. If the terminal is in the VDT state, the highest addressable line number for the terminal is determined by issuing an SVC to read the device characteristics of the terminal. (This information is stored in the variable VDTSIZ, and is referenced in other places by the Text Editor. The initialization phase is the only time the SVC is issued.) DLINES is set to the number of displayable lines. If the state is TTY, DLINES is set to 1. If the state is TTY and the device is a VDT, a message is displayed telling the user to change to VDT mode, and the Text Editor terminates. This is done because there are fundamental, unreconciled differences between the way the 733/820 DSRs and the VDT DSRs work. If the VDT DSRs are modified to operate in the TTY mode as the 733/820 DSRs work (currently, the VDT DSRs do not recognize TTY/VDT states), this restriction could be removed.

The strip carriage control flag, E$STRP, is cleared. This flag is used to limit to one the number of times an error message is generated while editing a file that contains undisplayable characters (for example, the carriage control characters of an ANSI print file).

The open file flags are cleared and the flag field of the input file IRB is set. Internal file position indicators are initialized.

When control is returned to E$EDIT, the routine S$PARM is called to obtain the access name of the input file, if one is provided. E$FLIO is called directly to open the input file if any and to create and open the MOD and TEXT temporary files. A conventional E$FLIO call is made to position each file at the beginning of the file. The edit in progress flag, EDTFLG, is set.

Initialization for an edit session is complete. The remaining processing of E$EDIT is executed each time it is called.

E$PRMS obtains the vertical and horizontal roll, right margin, scaling and tab stop parameters. E$FOPN gets the exclusive edit and length parameters. The roll and right margin values are stored in the resident data area as integers, and a tab stop bit map is built from the tab parameter values.

The value of the scaling parameter is checked and the following logic is executed if scaling is requested:

```
  IF VDT state
     THEN
        Set DLINES to VDTSIZ-1 (bottom line for scaling);
        Decrement roll by one;
        Set flag to reserve bottom display line for scale;
  ENDIF;
```

Note that during any edit session, scaling can be turned on and off as desired.

DISPAG is called to display the current page. The Text Editor is now prepared to accept user edit function selections.

Terminate Session.

SCI Command: QE

CODE: 9

PARMS:

Number	Definition
1	Abort?
2	Output file access name
3	Replace?
4	MOD list access name

The Quit Edit (QE) command terminates an edit session and is processed in the module E$QE. When the first PARM is NO, creation of the MERGE file is performed in the module E$FMRG. A rename SVC (Assign New Pathname operation of the I/O SVC) is issued in E$QE to change the name of the MERGE file to the specified output file name.

In the merge process, records are read from the MOD file to determine how the MERGE file is built. Records are copied from the input file or skipped (if deleted, changed, or inserted), and records from the TEXT file replace input file records for changes and insertions. If a MOD listing file is specified, the appropriate information, including proper carriage control, is written to it.

During terminate session processing, each of the three levels of I/O is used: services are requested through the control point E$FMNG, I/O routines in E$FLIO are called directly, and during the rename process, one direct I/O SVC is issued.

Upon successfully building (and possibly renaming) the new file, the Text Editor task is terminated by a call to E$WAIT, which calls S$TERM.

Recover Edit Session.

SCI Command: RE

CODE: 11

PARMS:

Number	Definition
1	Abort?
2	Output file access name
3	Replace?
4	MOD list access name

Processing for the Recover Edit (RE) command does what is necessary to allow a terminate session request to be processed. The purpose of the command is to allow recovery of data entered/changed during an edit session that is active when the system crashes, or is stopped.

The work files for that edit session are opened, and the pathname of the previously specified input file is read from the TEXT file header record. The input file, if any is opened.

The following variables are initialized as shown. These values are not necessarily what they are at the time the edit is interrupted, but they do allow session termination processing to complete.

* LSTINP=Number of lines in the input file minus one. This number is obtained when the input file characteristics are read.

* MODEOF=LSTINP+1. This is the entry number of the MOD file entry.

* E$NORC=LSTINP. In normal circumstances this value is a running count of input records processed. It is initialized as the number of lines in the input file, if any.

* HIMRN=>7FFF. This variable is the highest MOD file entry number. It is set to an arbitrarily high number.

* BGNREC=-1. This is an initializing value to ensure that all MOD entries are processed.

Following this initialization, control is transferred to terminate session processing in the module E$QE. The MERGE file is created and disposed of as indicated by the elements of the PARMS list.

As long as the MOD, TEXT and input files are readable, the edit activity is recovered, with the exception of any MOD entries (and any TEXT records that they reference) that are buffered, but not written to the MOD file at the time the system fails.

The MOD disk file is updated when the LINTBL does not contain all entries that are needed. The most damaging instance occurs when a previously empty file is being built and 24 lines (25 line VDT) have been entered when the system fails. LINTBL will, so far, have contained MOD entries for the entire file. No MOD file records will have been stored on disk. Recover edit, in this case, finds an empty MOD file and reconstructs a null file.

5.8.5.2 Independent Commands.

Modify Tabs (MT), Modify Roll (MR), Modify Horizontal Roll (MHR) and Modify Right Margin (MRM) commands are independent commands in the sense that there is no special processing done by the Text Editor when these commands are entered. In the command procedures shipped by Texas Instruments Incorporated, these commands only bid the Text Editor when an edit session is active. When the Text Editor is bid, a CODE value of 0 is used, so that the commands reactivate the session. If no edit is active, the command procedures set synonyms but do not bid the Text Editor.

5.8.5.3 Line and String Commands.

All command functions call E$EVAL to convert the line parameters into absolute, relative, or combination absolute/relative line numbers. If the absolute portion of the line number is zero, the default value of the current cursor line is assumed. E$EVAL does not contain the logic to process the special cases of BEGIN and END as line numbers. Each command processor takes action on these values before calling E$EVAL.

All command processors call E$POSF to position the file, based on the evaluation of line numbers. The MOD entry in LINTBL is altered, and the input file and TEXT file are read, as specified by the MOD file, to produce the text to be displayed at the terminal.

<u>Copy, Move, Delete Lines</u>.

PARMS and CODE:

Copy Lines (CL)	2	Start line, end line, insert after line
Move Lines (ML)	3	Start line, end line, insert after line
Delete Lines (DL)	4	Start line, end line

These three closely related commands are processed by code in the module E$CLML. The entry points for the Copy, Move and Delete Lines commands are ECL, EML and E$DL, respectively. The command flag is set to a unique value at each entry point and control is transferred to common code at E$LCOM.

Prior to positioning the file for processing, each line parameter is evaluated. If the parameter has an absolute component, an attempt is made to position the file. This determines if the specified line actually exists in the edited file. If one of the three parameters references a missing line, processing is terminated with an error message. If there is no absolute component, position in the file is moved the specified number of lines in the proper direction (or until the beginning-of-file or end-of-file is encountered). During this validation process, the number of records to be moved, copied, or deleted is counted. This information is used later to calculate the amount of memory required for a copy or move temporary buffer.

After the line numbers have been successfully validated and the file positioned, the following processing is done:

```
Position the file to the line specified by the first parameter;
IF this is a copy or a move
   THEN issue an SVC to get memory for temporary buffer space;
DO from start line to end line;
   Call E$FMNG to read the text of the line and page
    forward one line;
   IF Copy or move
      THEN Write the text of the line to the temporary buffer;
   IF Move or delete
      THEN Call E$FMNG to delete the line;
END;

IF Copy or move
   THEN
      Position file to the line specified by the third
       parameter;
      Insert temporary buffer into edit file, in reverse
       order (end line to start line);
      Issue an SVC to release the memory acquired for
       temporary storage.
ENDIF;
```

Insert File.

SCI Command: IF

CODE: 8

PARMS:

Number	Definition
1	File pathname
2	Insert after line

Processing for the Insert File (IF) command is in the module E$IF. S$PARM is called to obtain the access name of the file that is to be inserted. The file is opened by a call to E$OPEN (the standard I/O interface is not used because the file that is opened and read is not the file being edited). The file being edited is positioned by a call to E$POSF, which, in this case, fetches parameter number two and calls E$FLIO to read the record for that line. E$FMNG pages the file being edited forward one line and inserts the record before the current line. This process is repeated until an EOF is encountered in the file being inserted, at which time the inserted file is closed and insert file processing ends.

Save Lines.

SCI Command: SVL

CODE: 10

PARMS:

Number	Definition
1	Start line
2	End line
3	Save file pathname
4	Option(ADD,REPLACE,EXTEND)

Processing of the Save Lines (SVL) command is done in module E$SVL. The current file position is saved so that it can be restored later. The edited file is positioned at the beginning and ending line numbers in order to verify those values. E$SVL issues an assign LUNO (with autocreate) to the save file pathname. The file is opened and, if it was not created by the assign, the fourth parameter (option) is tested to determine what action to take regarding replacement. The ADD option stores the file if it does not currently exist. REPLACE saves the given lines whether the file exists or not, and EXTEND adds the lines to the end of the given file. An error condition is set and control returns to SCI for display of the error message.

Lines are copied from the edit file to the save file until the line count is exhausted. (The line count is set during positioning of the file at the beginning and ending line.) An EOF is written to the save file, it is closed and processing of the request ends.

<u>Show Line</u>.

SCI Command: SL

CODE: 1

PARMS:

Number	Definition
1	Line number

Processing of the Show Line (SL) command is controlled by E$SL. After the only parameter is obtained, the first character is compared with B for beginning-of-file and E for end-of-file. If not B or E, it is assumed to be a line number. The file is positioned using the appropriate call to E$FMNG. DISPAG displays the page.

<u>Find String</u>.

SCI Command: FS

CODE: 5

PARMS:

Number	Definition
1	Number of occurrences
2	Start column
3	End column
4	String

The parameters passed are stored in E$PRMB, a parameter text buffer. E$SPRM is called to obtain and validate the first four elements of the PARMS list.

The Find String (FS) command is processed in module E$FS. The text of the cursor line is read and E$MTCH is called to search the line for a match to the string. If one is found, the occurrence count is decremented. If the occurrence count is zero, the command is terminated, leaving the cursor at the beginning of the (last) line on which a match is found. If a match is not found, or if one is found but the occurrence count is not yet decremented to zero, E$FMNG is called to page forward one line and the next line is processed as described above. Lines are read until the EOF record is encountered or the cursor is positioned to the first character of the string found until the occurrence count is decremented to zero.

Replace String.

SCI Command: RS

CODE: 6

PARMS:

Number	Definition
1	Number of occurrences
2	Start column
3	End column
4	String
5	Replacement string

The parameters passed are stored in E$PRMB, a parameter text buffer. E$SPRM is called to obtain and validate the first four elements of the PARMS list.

Processing for the Replace String (RS) command is done in the module E$RS.

The fifth parameter, the replacement string, is obtained by a direct call to S$PARM.

In E$SREP, the text for the line denoted by the cursor is read by E$FMNG. E$MTCH is called to search the line for an occurrence of the specified string. If one is found, the characters to the right are shifted appropriately to make room for the replacement string, if any, which is inserted into the line. If the replacement string is shorter, blanks are inserted from the right margin to fill the line. An error is generated if the starting column plus the replacement string length is greater than the right margin. E$FMNG is called to write a change record for the altered line, and the starting column is updated for the next call to E$MTCH. The same line is examined for a match until no match is found, at which time E$FMNG is called to page forward one line, to make available the next line to be examined. This operation continues until the occurrence count is zero or until the EOF record is read. If there is no input string, all characters between the start and end column are assumed to match, and are replaced by the replacement string.

The number of blanks that can be appended to the right of the line is limited to 80. Otherwise, an attempt to delete blanks from the entire line would cause an endless loop of delete blank, shift left and pad with a blank.

Delete String.

SCI Command: DS

CODE: 7

PARMS:

Number	Definition
1	Number of occurrences
2	Start column
3	End column
4	String

The parameters passed are stored in E$PRMB, a parameter text buffer. E$SPRM is called to obtain and validate the first four elements of the PARMS list.

Processing for the Delete String (DS) command begins in the module E$DS. The process is the same as for the Replace String (RS) command, except that the replacement string is null, that is, the length of the replacement string is zero. Control is transferred to E$SREP where the search and deletion takes place.

5.9 ERROR PROCESSING

The Text Editor uses the same S$ error reporting facilities used by SCI.

The SEC macro is used to generate code to call S$XFER. All error conditions are set using constants of the format E$Exxx. These constants are defined in module E$ERRORS. A cross-reference listing for the Text Editor shows the modules in which an error condition is set.

5.10 MODIFYING THE TEXT EDITOR

Care must be exercised when modifying the module E$QE, which contains the code to rename the MERGE file. At any time during this process, the edited and/or original input file must remain recoverable in case of a system crash.

In order to allow sharing of the code by more than one terminal, any volatile code (such as calculated branch instructions) must be in a DSEG.

5.11 INTERNATIONALIZATION

There is no embedded text in the Text Editor code.

SECTION 6

SYSTEM CONFIGURATION UTILITY

6.1 OVERVIEW

The system configuration utility (SCU) allows a user to modify the image of a DNOS operating system as created during system generation (sysgen), or to modify basic attributes of the currently executing system, without having to reexecute the System Generation utility. The capabilities provided by SCU include:

* Listing the current device configuration
* Modifying, adding, or deleting devices. The same attributes of a device that can be specified during sysgen can be modified through SCU.
* Showing and/or modifying the country code
* Modifying the sizes of various system table areas
* Modifying various scheduler and swapping parameters
* Modifying the current state of an existing device
* Initializing the system log

Some of the previously listed capabilities are not available when modifying the currently executing system (for example, when changing, deleting, or adding devices or changing system table area sizes).

SCU is written in Pascal, with calls to assembly language routines located in the system root. This section describes the logic flow and processing performed by SCU.

6.2 STRUCTURE

The configuration utility is a DNOS system task with overlays.

6.2.1 Address Space.

SCU's logical address space is slightly different when modifying disk images and when modifying the running system.

When modifying the running system, SCU has the system root in the first segment, and, since SCU is a system task, the user's SCI job communication area (JCA) in the second segment. The SCU task area, including overlays, is in the third segment.

When modifying a disk image, the system root occupies SCU's first two map segments. The DNOS system root is installed as two separate segments on the kernel program file. (During the initial program load (IPL) sequence, the system loader coalesces the two root segments into one contiguous segment, the system root.) When SCU maps these disk image segments into memory, they are treated by the Segment Manager as two distinct segments, although SCU maps them such that all logical addresses in the root are valid.

Figure 6-1 shows the different logical address spaces in which SCU may operate.

6.2.2 Special Features.

SCU uses the updateable program file processing of the Segment Manager to rewrite the disk image of an operating system.

Most changes to the devices in a system configuration are accomplished by first deleting the old device definition and then adding the new one. The exception is the modification of the state of a device. This does not require any change in system data structure chaining, and therefore can be handled without the delete and add operation.

When a device is deleted from the system PDT list, SCU calls the nucleus support table management routines NFGTA and NFRTA to compress table memory.

The SCU program operates in either interactive or batch mode. The Modify Device Configuration (MDC) command procedure structure is not, however, executable in batch.

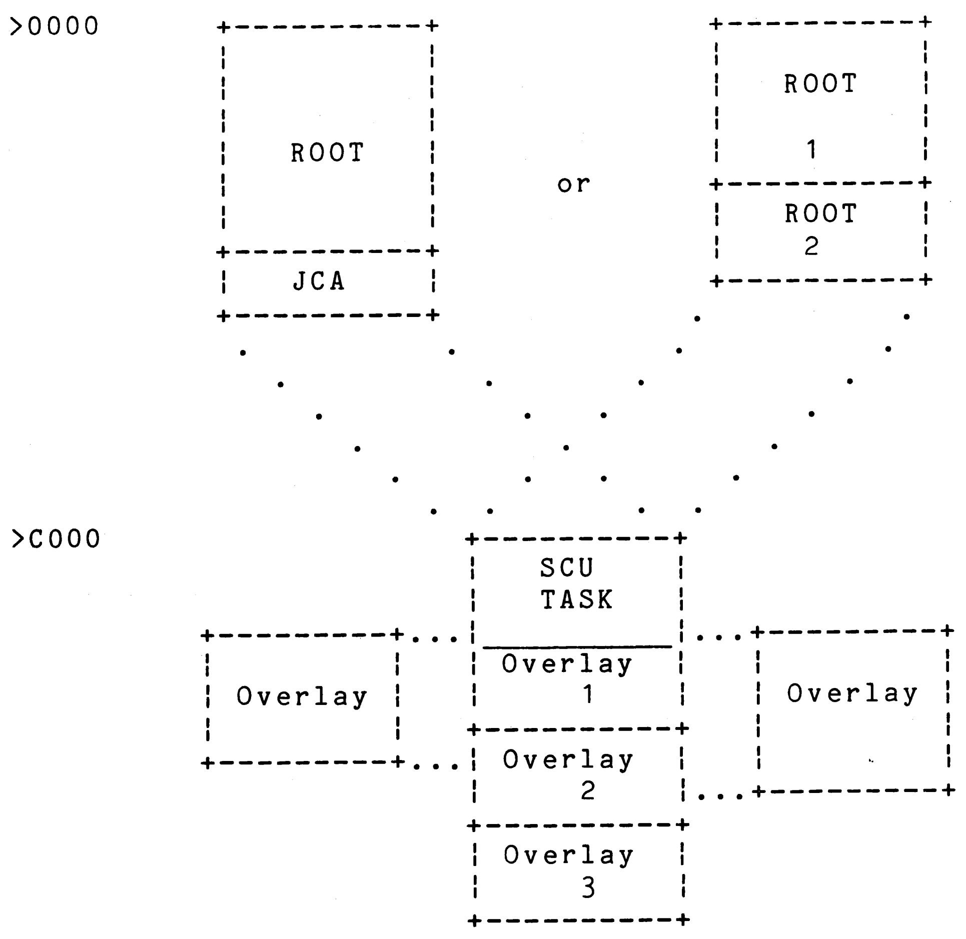


Figure 6-1 SCU Address Space

6.2.3 Overlays.

There are seventeen SCU overlays as shown in Table 6-1. The program file IDs are defined in the Pascal template SCUCONS. A maximum of three overlays can be resident in the SCU address space. Figure 6-2 shows the calling structure for overlays.

Table 6-1 SCU Overlays

Overlay Name	Function(s)
SCUINIT	Initializes and terminates SCU session Returns Device Parameters Processes Modify Country Code (MCC) command
SCUDATA	Builds the SCU internal data base from system tables
SCUDEV	Processes a Modify Device Configuration (MDC) command
SCULDC	Processes List Device Configuration (LDC) command
SCUADD	Adds a device to the system configuration
SCUPDT	Builds the appropriate system data structures for a device being added
SCUPD1	Fills in certain extensions to the PDT
SCUPD2	Fills in certain extensions to the PDT
SCUDSR	Installs a DSR for a device being added
SCUAINT	Adds Single and multiple device interrupt tables
SCUAEXP	Adds Expansion chassis interrupt tables
SCUAMUX	Adds MUX board interrupt tables
SCUDEL	Deletes a device
SCUMDS	Processes Modify Device State (MDS) command
SCUMISC	Processes Modify System Table Sizes (MST) command Processes Initialize System Log (ISL) command
SCUMSP	Processes Modify Scheduler/Swap Parameters (MSP) command
SCUNAME	Allocates names to devices

6.3 FLOW OF CONTROL

SCU is bid by SCI. It processes the request, using the PARMS list passed by the command procedure, then calls R$WAIT to suspend SCU and to reactivate SCI. When SCU is invoked to process a request and no session is active (SCU maintains an internal flag to indicate whether a session is in progress), SCU terminates through R$TERM instead of R$WAIT. SCU also terminates through R$TERM after processing a request to terminate the session (QSCU command).

```
                               CUMAIN
                                 !
            +---------+----------+----------+----------+
            !         !          !          !          !
        SCUINIT     SCUDEV     SCUADD     SCUMSP    SCUMISC
                      !          !
     +-------+-------+-------+   !
     !       !       !       !   !
   SCULDC  SCUDATA  SCUDEL SCUMDS!
                                 !
           +--------+--------+---+----+--------+--------+
           !        !        !        !        !        !
         SCUPDT   SCUPD1   SCUPD2   SCUNAME  SCUDSR   SCUAINT
                                                        !
                                                 +-----------+
                                                 !           !
                                              SCUAMUX     SCUAEXP
```

Figure 6-2 Calling Structure for SCU Overlays

6.3.1 Invoking SCU.

SCI invokes SCU by .RBID primitive. In general, information is passed from SCI to SCU by parameters on the RBID statement. The exception is the request to modify system parameters. Parameters are passed to SCU in synonyms for that process. Information is always passed from SCU to SCI by synonyms.

The PARMS list varies, according to the service requested, but the first three elements are always the same, as follows:

PARM No.	Definition
1	Pascal stack parameter
2	Pascal heap parameter
3	Opcode for request

The contents of the fourth through last elements of the PARMS list depend on the opcode. Variations of the PARMS list with specific opcodes are discussed, with the detailed design of the code, in the following paragraphs.

SCU does not make use of the CODE parameter on a bid statement.

6.3.2 Initialization.

The initialization of SCU is done in the procedure CUINIT, which is called in CUMAIN. This code is executed only on the initial bid of SCU by the XSCU command, as opposed to subsequent RBIDs. The process consists of setting up the logical address space of the utility for the system being modified and initializing session and error variables used by SCU.

6.3.3 Main Program.

The driver routine for SCU, CUMAIN, has two major phases of execution: initializing the memory-resident data base, and processing changes to the operating system being modified. CUMAIN contains a loop that gets a new opcode to be processed, calls the appropriate opcode processor, and calls R$WAIT to be suspended until SCI restarts SCU with the next request.

At the beginning of the opcode processing loop, the SCU variable REINIT is tested to determine whether the device definition data base needs to be rebuilt. This could be necessitated by changing, deleting, or adding a device. If it does need to be rebuilt, CUDATA is called.

CUDATA executes in three stages, as follows:

1. Deletes all current device definitions in the SCU internal data structure.

2. Reads operating system tables to determine which devices are defined, and the associated interrupt level, expansion chassis, and expansion position for each device. This information is stored in an SCU data structure, the device definition list.

3. Scans the PDT list, adding entries to the device definition list for devices with no interrupt.

Building the device definition list involves processing the interrupt trap table located in memory locations >0000 through >003F and following the appropriate chain of system data structures until the physical device table (PDT) associated with each device is located. The data structures that make up this chain are discussed in the subsequent paragraphs.

If more than one device is defined at a given interrupt level, CUDATA processes every entry in the multiple-interrupt decoder table, handling each device interrupt vector the same way as in the single device case.

If an expansion chassis is defined at an interrupt level, each position for the chassis is processed the same way as in the single device case.

Also, if one or more asynchronous multiplexor (MUX) boards is defined at an interrupt level all devices on each board are processed as in the single device case.

6.3.4 Termination.

SCU is designed to protect the integrity of the home file operating system in the event of a system failure. That is, the root segments are not marked modified during an active SCU session, so that in the event of an irrecoverable error, the Segment Manager does not destroy the originals on disk. The Segment Manager SVC that marks the memory-resident segments modified is issued during normal termination of an SCU session.

NOTE

> This scheme only works as long as updateable segments are swapped out (rather than rewritten to disk) when SCU is swapped out. Should the swapping of updateable segments by the Segment Manager change in this regard, SCU will have to be modified, perhaps drastically.

SCU sets the $$CA (configuration utility active) synonym. During a session, the synonym is set to either YES or NO, and at the end of the session, the synonym is deleted. The synonym is set to YES before R$WAIT is called to return control to SCI. However, the synonym is set to NO during actual execution of SCU, so that, in case of an abnormal termination, the synonym's meaning remains correct.

6.3.5 Error Processing.

Errors detected in SCU code are of two types: errors that cause abnormal termination of SCU, and warnings. Errors that cause termination are generated only by S$ routines, and are handled by the common utility routine, UTCHEK. Warnings are caused by errors detected by SCU code (such as NO SUCH DEVICE), and cause SCU to abort the request being processed and return an error to the user through R$WAIT. SCU remains active, and any session in progress remains in progress, although the warning message may advise the user to abort the session.

Irrecoverable errors are not reported up the calling chain to the SCU main driver; they cause immediate task termination. Other errors are reported to the main driver eventually by setting four Pascal common error variables, $$CC, $$VT, $$MN, and $$ES. The values of these four variables are passed as the parameters to R$WAIT at the end of the major loop in the SCU main driver, CUMAIN. R$WAIT transfers the value of each variable to a synonym with the same name, and suspends SCU.

6.4 DATA STRUCTURES

SCU accesses several operating system data structures. The only data structures maintained by the program for its exclusive use other than data structures used by the Pascal run time, are the device definitions list, the device map file array (DEVMAP), error variables, and some global flags.

System data structures as well as SCU internal data structures and variables are discussed in the subsequent paragraphs.

6.4.1 Interrupt Trap Table and Supporting Structures.

In order to access information about devices currently defined on the system being modified, SCU must read the PDT (and any associated extension data structures) associated with each device. Even though the PDT associated with each device may have an extension, in the remainder of this section, the set of one or more structures is called the PDT. Any reference to a PDT includes a reference to the appropriate set of data structures, depending on the device.

For each device that is capable of generating an interrupt, there exists, in operating system tables, a device interrupt vector data structure. A device interrupt vector is a six-byte data structure that contains:

* WP. A pointer to the interrupt workspace (either a PDT, a keyboard status block (KSB), or a multi-unit workspace (MUW)) for the device. The format of this data structure, as well as all extension data structures except the MUW, is shown in the Data Structure Pictures section of <u>DNOS System Design Document</u>. The structure of the MUW is shown in Figure 6-3. The PDT may also have extension data structures of the following types:

  - Keyboard status block (KSB)

  - Disk PDT extension (DPD)

  - Line printer PDT extension (LPD)

- Extension for a Terminal with a Keyboard (XTK)
- Device Information Block for TPD DSR (DSTDIB)
- Local ASYNCH Extension to the PDT (DSALLLEX)
- Multi-unit workspace (MUW). This structure is a 16-register workspace, and is associated with a controller that interfaces with as many as four devices. Each device associated with the controller has a PDT, but it is the controller that generates an interrupt to the operating system. Figure 6-3 shows the register assignments in the MUW.

* PC. Address of the DSR for the device. Because of the manner in which DSRs are installed in DNOS, this address is always >C000. This fact is not, however, exploited in SCU code.

* MAP. Address of the map file for the device

```
        +------------------------------------------------+
R0      | PDT0 -  Pointer to PDT workspace for first     |
        |         device associated with controller      |
        +------------------------------------------------+
R1      | PDT1*                                          |
        +------------------------------------------------+
R2      | PDT2*                                          |
        +------------------------------------------------+
R3      | PDT3*                                          |
        +------------------------------------------------+
R4      | Flags                                          |
        +------------------------------------------------+
        \                                                /
        /                                                \
        +------------------------------------------------+
R12     |  TILINE address of the controller              |
        +------------------------------------------------+
R13     |  Return context:  Workspace pointer            |
        +------------------------------------------------+
R14     |                   Program counter              |
        +------------------------------------------------+
R15     |                   Status register              |
        +------------------------------------------------+
```

*   Reserved for pointers to PDT workspaces for a maximum of four devices.

Figure 6-3 Multi-Unit Workspace Structure

A unique device interrupt vector for each device is defined to the operating system. The following paragraphs describe the system data structures through which the vectors are located.

The operating system interrupt trap table contains the initial pointers for all devices. This structure is stored in >0000 through >003F of SCU's logical address space. For each of the 16 interrupt levels (0 through 15), the interrupt trap table contains a workspace pointer and a program counter.

The program counter for each device interrupt level is one of four entry points into the interrupt processor. The following addresses are defined in the system data structure NFPTR:

* PCSPTR - Entry point for processing if a single device is defined at this interrupt level
* PCMPTR - Entry point for processing if multiple devices are defined at this interrupt level
* PCEPTR - Entry point for processing if an expansion chassis is defined at this interrupt level
* PCAPTR - Entry point for processing if a MUX board is defined at this interrupt level.
* ILLPC - Entry point for processing if no device is defined at this interrupt level

The program counter in the interrupt trap table is compared with those global constants to determine what kind of structures if any must be examined in order to find the associated device interrupt vector(s).

If the program counter address indicates a single device, the WP value is the address of a workspace in which registers eight, nine and ten form a device interrupt vector, as shown in Figure 6-4.

```
        Trap
       Table
  0 +---------+                                      Device
    |    .    |                             +-------->  PDT
    |    .    |                             |
    |    .    |          Workspace          |
    +---------+         +-----------+       |
    |   WP --+-------->/           /        |
    +---------+         +-----------+       |
    | PCSPTR  |      R8 |Device WP  |---+     <--+  Device
    +---------+         +-----------+            |
    |    .    |      R9 |Device PC  |            |  Interrupt
    |    .    |         +-----------+            |
    |    .    |     R10 |Device MAP |          <--+  Vector
    |    .    |         +-----------+
    |    .    |         /           /
    |    .    |         +-----------+
 >40+---------+
```

Figure 6-4 Single-Interrupt Decoder Data Structures

If the interrupt trap table program counter is PCMPTR, multiple devices are defined at the interrupt level. The WP value points to a workspace where register nine points to a multiple-interrupt decoder table. A multiple-interrupt decoder table is an array of four-byte entries. The first two bytes are a communications register unit (CRU) bit address. The third and fourth bytes are a pointer to a device interrupt vector as shown in Figure 6-5. Each device defined at that interrupt level has one entry in the table. A CRU bit address of zero denotes the end of the table.

```
      Trap
     Table                Multiple-interrupt Decoder Table
 0 +--------+                   +--------------------+
   |   .    |             +--->|   CRU  | vector    |
   |   .    |             |    +--------+----------+
   |   .    |    Workspace|    |   CRU  | vector --+--+
   +--------+    +------+ |    +--------+----------+  |
   |   WP --+--->/      / |    /        /          /  |
   +--------+    +------+ |    +--------+----------+  |
   | PCMPTR |    |  R9 -+-+    |   0    |    0     |  |
   +--------+    +------+      +--------+----------+  |
   |   .    |    |      |                             |
   |   .    |    /      /    +-------------------------+
   |   .    |    +------+    | Device Interrupt Vector
   |   .    |                |      +----------+
   |   .    |                +----->|Device WP |
   |   .    |                       +----------+
>40+--------+                       |Device PC |
                                    +----------+
                                    |Device MAP|
                                    +----------+
```

Figure 6-5 Multiple-Interrupt Decoder Data Structures

If the program counter in the interrupt trap table is PCEPTR, an expansion chassis is defined at the interrupt level. The chain of data structures for an expansion chassis definition is shown in Figure 6-6. WP points to a workspace where the value in register twelve indicates which expansion card is defined at that level. A value of >1F00 indicates card one (chassis 1 through 4) while >1F20 indicates card two (chassis 5 through 7). Up to seven expansion chassis may be defined in a DNOS system. ETAB is an operating system table with seven entries, two bytes for each possible expansion chassis. Each entry contains either a pointer to a chassis position table or zero (chassis undefined). The address of ETAB is in the ETBPTR field of NFPTR.

A chassis position table contains 24 entries, four bytes for each possible position (0 - 23) on an expansion chassis. Two bytes are a flag that indicates single or multiple devices at the position. The remaining bytes point either to a multiple-interrupt decoder table as shown in Figure 6-5 or to a device interrupt vector for a single device as shown in Figure 6-6), or are zero (position undefined). Figure 6-6 shows the path from the interrupt trap table to each of the devices associated with an expansion chassis definition.

```
                                   ETAB          Chassis
                                 +-------+     Position Table
                               ->|Chas 1+---+  +-----+-----+
         Trap                  . +-------+  +->|Pos 0|Flag*|
        Table                  . /       /     +-----+-----+
   0 +----------+              . +-------+     |Pos 1|Flag*|
     |    .     |              . |Chas 4/      +-----+-----+
     |    .     |              . +-------+     |Pos 2|Flag*|
     |    .     |    Workspace . /       /     +-----+-----+
     +----------+    +--------+ . +-------+    /     /     /
     |   WP  --+---->/        / . |Chas 7|     |-----+-----+
     +----------+    +--------+ . +-------+    |Pos23|Flag*|--+
     |   PCE    |  R12| >1F00 |..              +-----+-----+  |
     +----------+    +--------+                               |
     |    .     |    |        |                               |
     |    .     |    /        /          +--------------------------+
     |    .     |    +--------+          |    Device Interrupt Vector
     |    .     |                        |         +-----------+
     |    .     |                        +-------->| Device WP |
     |    .     |                                  +-----------+
  >40+----------+                                  | Device PC |
                                                   +-----------+
                                                   | Device MAP|
                                                   +-----------+
```

* Flag indicates single/multiple device(s) associated with chassis position. In this example, a single-device definition is shown.

Figure 6-6 Expansion Chassis Decoder Data Structures

If the program counter in the interrupt trap table is PCAPTR, a MUX card is defined at the interrupt level. The chain of data structures for a MUX card definition is shown in Figure 6-7. WP points to a workspace where the value in register ten points four bytes proceeding a MUX board table. A MUX board table contains a three word entry for each MUX card defined at this interrupt. The first word of an entry contains the TILINE address of the card. The second word contains a pointer to the channel table associated with the board, and the third word is unused in DNOS. The table is terminated by a word of binary zeros. Each MUX board must have one, and only one, four-entry channel table, the entries corresponding to channels zero through three, respectively, on the board. Each entry contains a standard DSR entry vector plus an unused word.

```
                                   MUX Board Table
         Trap                   +-->+----+----+
        Table                   |   :         :
     0 +-------+                |   :----+----:
       |   :   |                |   :         :
       |   :   |    Workspace   |   +----+----+
       +-------+    +-------+   |   | TILINE  |
       |  WP --|--> /       /   |   |Address 1|
       +-------+    +-------+   |   +----+----+    Board 1
       |  PCA  | R10|       |---+   | Channel |
       +-------+    +-------+    +--|Table PTR|
       |   :   |    |       |    |  +----+----+
       |   :   |    /       /    |  |   >18   |
    >40+-------+    +-------+    |  +----+----+
                                 |  |    :    |
                                 |  |    :    |
             Channel Table 1     |  +---------+
               +----------+      |  | TILINE  |
               |Device WP | <----+  |Address N|
    Channel    +----------+         +---------+    Board N
      # 0      |Device PC |         | Channel |
               +----------+         |Table PTR|---+
               |Device MAP|         +---------+   |
               +----------+         |   >18   |   |  Channel Table N
               |    0     |         +---------+   |    +----------+
               +----------+         |    0    |   +--->|          |
               |    :     |         +---------+        |          |
               +----------+                          |          |
               |Device WP |                          |          |
    Channel    +----------+                          |          |
      # 3      |Device PC |                          +----------+
               +----------+
               |Device MAP|
               +----------+
               |    0     |
               +----------+
```

Figure 6-7 MUX Interrupt Decoder Data Structures

6.4.2 System Common Area.

System common areas defined in NFDATA, NFPTR, NFJOBC and NFCLKD are accessed and modified by SCU. The Data Structure Pictures section of the DNOS System Design Document contains the detailed formats of system common areas.

6.4.3 SCU Internal Data Structures and Variables.

SCU maintains a linked list of Pascal records that contain device definitions. Each device definition contains information condensed from the chain of structures associated with the device, and from the PDT for that device. The following information is kept in each record of the list:

* CRU or TILINE address

* Device PDT address. If the device is a controller, as many as four device PDT addresses are saved.

* Interrupt level, chassis number, and chassis position

* Number of units

* Interrupt workspace address

The internal list is rebuilt at the beginning of the major loop in CUMAIN if the device configuration was changed by the last command. The need to rebuild the list is indicated by a global flag, REINIT.

The global flags, SESSION and RUNNING_SYSTEM, indicate the current SCU environment (that is, whether or not a session is in progress, and whether or not the running system or a disk image is being modified).

6.4.4 Synonyms.

Seventeen synonyms are used by SCU. The names are as follows:

$CU1, $CU2, $CU3, $CU4, $CU5, $CU6, $CU7, . . .$CU17

This set of synonyms is used to pass information between SCU and SCI. The data contained in each synonym is dynamic, depending on which request is being processed by SCU. A mapping scheme that localizes assignment of synonyms for data transfer is used in most of SCU (synonym names are hard coded in the procedures CURISL and CUISL).

Mapping variables are defined in SCUCONS (in the TEMPLATE.PTABLE directory) These variables are of the following format:

$$name

where:

name is a meaningful character string -- it is either descriptive of the information to be mapped with the variable (CHAS for chassis, POS for position), or it is the exact name of a system variable from NFDATA or NFCLD.

Each mapping variable is given an integer value between one and seventeen. The mapping is effected in SCU code by assigning values to synonyms whose names are constructed as follows:

$CU@$$name.

As an example, suppose the synonym mapping variable $$MEMTIC has a value of five. In SCU code, the value of the NFDATA variable MEMTIC is passed back to SCI as the value of the synonym $CU5.

This synonym mapping scheme centralizes assignment of synonyms for data transfer between SCI and SCU. Only the $$name definitions in SCUCONS need be changed to alter the mapping.

The following synonyms are set by SCU code. Their names are hard coded.

* $$CA - Indicates whether an SCU session is currently active.

* $CU$RS - Indicates whether the running system or a disk image is being modified

* $TYP - Indicates device type. When SCU processes a request to return the characteristics of a device with a particular name, $TYP is set to a value that indicates the type device associated with that name. (See Table 6-3 for the values $TYP is assigned.)

6.5 DETAILED DESIGN

Each of the requests recognized by SCU is discussed in the following paragraphs. The opcode, overlay(s) and PARMS list are defined. Processing is discussed in general terms.

When SCU is invoked, the first three elements of the PARMS list are always the same -- stack, heap and opcode. Values of 500 for stack and 800 for heap are sufficient for the configuration utility. Table 6-2 summarizes the opcode values used by SCU. Only the fourth and subsequent elements of the PARMS list are discussed with each specific request.

Table 6-2 SCU Opcodes

Opcode	Function
0	Initiates an SCU session
1	Lists current device configuration
2	Requests device parameters
3	Modifies an existing device
4	Adds a device
5	Deletes a device
6	Shows country code
7	Modifies country code
8	Requests system table area sizes
9	Modifies system table area sizes
10	Requests system log parameters
11	Modifies system log parameters
12	Terminates SCU session
13	Requests system parameters - screen 1
14	Requests system parameters - screen 2
15	Requests system parameters - screen 3
16	Modifies system parameters - screen 1
17	Modifies system parameters - screen 2
18	Modifies system parameters - screen 3
19	Modifies device state
20	Requests system parameters - screen 4
21	Modifies system parameters - screen 4

In the following discussion, elements passed on the PARMS list and in SCU synonyms are defined as they appear in the DNOS System Command Interpreter (SCI) Reference Manual. Refer to the discussion of the referenced SCI command for additional information about the parameters.

6.5.1 Initiate SCU Session.

SCI Command: XSCU

Opcode: 0

Overlay: SCUINIT

PARMS:

One additional PARM must be passed. It is the name of the system to be configured.

Initialization of an SCU session is done in the procedure CUINIT which is called by CUMAIN. After initializing SCU variables, CUINIT is called to map into memory the two system root segments for the system being modified. This is done by assigning a LUNO to the program file containing the system to be modified, then issuing two change segment SVCs and a rebias segment SVC to map the root.

Following this initialization, control drops into the major loop, where the internal device definition list is built, and R$WAIT is called to suspend SCU until the next request is ready to be processed.

6.5.2 List Device Configuration.

SCI Command: LDC

Opcode: 1

Overlay: SCULDC, SCUDEV

PARMS:

The fourth element on the PARMS list must be the name of the output listing file.

CULDC is called to format a report from the SCU internal device definition data base. The report is written to the specified output destination by calls to R$WRIT, the Pascal interface routine to S$WRIT.

6.5.3 Device Characteristics.

Four requests are concerned with devices -- return device parameters, change device parameters, add device and delete device.

6.5.3.1 Return Device Parameters.

SCI Command: MDC

Opcode: 2

Overlay: SCUMDC

PARMS:

The fourth element of the PARMS list is the name of the device.

SCU returns the current values for device characteristics. Routine CURDP searches the device definition data base for a device with the specified name, and sets, according to the device type, a subset of the following synonyms:

* $CU$$DEVNAME - Name of the device

* $CU$$CRU - CRU or TILINE address of the device
* $CU$$EXTENDED - YES/NO line printer has extended character set
* $CU$$INT - Interrupt level
* $CU$$CHAS - Chassis number
* $CU$$POS - Expansion chassis position
* $CU$$QUEUE - Length of the character queue of a terminal
* $CU$$MODE - Print mode for line printer (S for serial, P for parallel)
* $CU$$DRIVES - Number of devices on a controller
* $CU$$DEFAULT - Default physical record size for disk
* $CU$$INTERFACE - Interface type
* $CU$$SWITCHED - Phone line switched YES/NO
* $CU$$BAUD - Baud rate of the station
* $CU$$ACU - Automatic call unit YES/NO
* $CU$$ACRU - Address of the automatic call unit
* $CU$$PRINTER - Associated line printer for the 940 YES/NO
* $CU$$MODEM - Full duplex modem YES/NO
* $CU$$ECHO - Echo keystrokes YES/NO
* $CU$$TYPE - KSR terminal type
* $CU$$CDT - CDT number
* $CU$$CDE - CDE mask

The synonym $TYP is set. The values assigned this synonym are determined by the operating system LDT template, as shown in Table 6-3. All values are given in hexadecimal.

Table 6-3 $TYP Values

$TYP Value	Device type
2	KSR
3	ASR
7	Disk drive
8	Magnetic tape drive
9	TPD device
A	911 VDT
B	Serial line printer
C	Parallel line printer
10	Card reader
11	940 VDT
12	931 VDT

6.5.3.2 Change Device.

SCI Command: MDC

Opcode: 3

Overlays: SCUDEV, SCUADD, SCUDEL, SCUPDT, SCUDSR, SCUPD1, SCUPD2, SCUAINT, SCUAEXP, SCUAMUX

PARMS:

Depending on device type, values must be supplied for a subset of the following PARMS list. A device is changed via three operations. The parameters associated with the old device are first retrieved. The device is then deleted. Finally, the old parameters together with any modifications to those parameters are used to add a new device.

REINIT is set to cause the device definition list to be rebuilt by CUDATA the next time through the major loop of CUMAIN.

PARM No.	Definition
4	Device name
5	CRU or TILINE address
6	Interrupt level
7	Expansion chassis
8	Expansion position
9	Device type
10	Drives
11	Print mode (serial or parallel)
12	Print width
13	Extended character set?
14	Time-out
15	Opens validated?
16	Character queue
17	Cassette time-out
18	Cassette opens validated?
19	Default record size
20	KSR type
21	Interface
22	Switched
23	Baud
24	ACU
25	ACU CRU
26	Echo
27	Full duplex
28	Associated printer
30	CDT number
31	CDE mask
32	Channel number

6.5.3.3 Add Device.

SCI Command: MDC

Opcode: 4

Overlays: SCUADD, SCUPDT, SCUDSR, SCUPD1, SCUPD2, SCUAINT, SCUAEXP, SCUAMUX

PARMS:

Same as change device.

CUAD is called to build the device record.

CUADD is called to allocate operating system table area for a new PDT. CUDSR is called to install the DSR, if necessary, and to link the PDT into the operating system PDT list. CUAINT is called to add the device at the specified interrupt level.

This process is more complex when the kind of data structure is changed (for example, the addition of the second device at an interrupt level changes the chain from the single-device structure to a set of multiple-device structures).

REINIT is set to cause the device definition list to be rebuilt by CUDATA the next time through the major loop of CUMAIN.

6.5.3.4 Delete Device.

SCI Command: MDC

Opcode:5

Overlays: SCUDEV, SCUDEL

PARMS:

The fourth element of the PARMS list is the name of the device.

CUMAIN calls CUDD to scan the device definition data base for a device that has the specified name. If one is found, IN_USE is called to determine whether the device may be deleted. It may be deleted if no LUNOs are assigned, and it is not an installed disk. If the device cannot be deleted, processing of the request is aborted. If it is acceptable to delete the device, CUDD calls CUDINT to delete that device from the interrupt trap table, then unchains the PDT for that device from the system PDT list and releases that memory to the system table area.

6.5.4 Show Country Code.

SCI Command: SCC

Opcode: 6

Overlay: SCUINIT

PARMS:

No additional input is required.

The current value of the country code, from the system table NFDATA, is returned to SCI by the routine CUSCC in the module CUCC.

The value returned in the synonym $CU$$COUNTRY is an integer. All mapping of the country code to country name is done in the command procedure.

6.5.5 Modify Country Code.

SCI Command: MCC

Opcode: 7

Overlay: SCUINIT

PARMS:

The fourth PARM is the new country code.

When the country code is changed, CUMCC, in the module CUCC, is called to replace the old value in NFDATA with the new one.

6.5.6 Show System Table Sizes.

SCI Command: MST

Opcode: 8

Overlay: SCUMISC

PARMS:

No additional input is required.

CURSTS processes this request and sets the following synonyms:

* $CU$$STA - Size of the system table area (STA). This is calculated from values in the STA overhead data structure, STAEND-STARES, which is the ending address minus the beginning address.

* $CU$$SMT - Size of the Segment Manager table area

* $CU$$FMT - Size of the File Manager table area

* $CU$$BTA - Size of the buffer table area (BTA), BTALEN in the data structure NFDATA

* $CU$$BTAMAX - Maximum size of BTA, BTAMAX in NFDATA

* $CU$$STAMAX - Maximum size of STA. This value is calculated using MAXSIZE (an SCU constant for maximum JCA/special table area size) and SYSTAB (from NFPTR, the address of the beginning of the system table area). The calculated value is calculated as follows:

* $CU$$SYS_JCA - System JCA size

STAMAX = >C000 - MAXSIZE - SYSTAB.

Calculating the Segment Manager and File Manager table areas involves reading segment status blocks (SSBs) in the STA. Refer to the <u>DNOS System Design Document</u> for details of SSBs. The lengths of segments belonging to each manager are summed to produce the total size of the tables allocated to the Segment Manager and to the File Manager. Pointers in NFPTR are used to access the appropriate SSB(s).

6.5.7 Modify System Table Area Sizes.

SCI Command: MST

Opcode: 9

Overlays: SCUMISC

PARMS:

PARM No.	Definition
4	Maximum job communication area (JCA)
5	New size for system table area
6	New size for segment manager tables
7	New size for file manager tables
8	New size for buffer tables
9	New size for system job communication area (JCA)

CUMAIN calls CUMSTS to process the values passed in the PARMS list. A new value for RELOCA, the relocation value used by the loader, is calculated and stored in NFDATA. The new value is the difference between the user-specified table area size and the current value of JCASTR (beginning address of the JCA), rounded to a beet boundary. The BTA size is changed by putting the new value in BTALEN in NFDATA.

Altering the size of the Segment Manager and the File Manager table areas may involve making changes in the STA and in NFPTR. The SSBs associated with the segments are in the STA. The maximum length of a segment represented by one SSB is >3000. If the total size of an area decreases, one or more SSBs may have to be deleted. If the area increases in size, new SSBs may need to be built. The building and destroying of special table area SSBs and updating the appropriate pointers in NFPTR are handled by the routine MOD_SSBS.

6.5.8 Show System Log.

SCI Command: ISL

Opcode: 10

Overlay: SCUMISC

PARMS:

No additional input is required.

This request is processed by the procedure CURISL, which returns current values from the system log for those items a user is allowed to change.

The following information is returned through SCU synonyms:

* $CU1 - Attention device name
* $CU2 - Log device name
* $CU3 - Task ID of system log processor
* $CU4 - Task ID of user log processor
* $CU5 - Number of records in each log file

The synonym assignments are hard coded in CURISL.

6.5.9 Initialize System Log.

SCI Command: ISL

Opcode: 11

Overlay: SCUMISC

PARMS:

PARM No.	Definition
4	Logging device
5	Attention device
6	System log processor task ID
7	User log processor task ID
8	Recreate file?
9	Allocation

Procedure CUISL is called to process most of the values by setting flags or changing values in the log processor common area, LGLCOM. If the user has specified that the log files are to be recreated, CUISL bids the log file recreate task, LGRCRT, in the system job. When LGRCRT terminates, CUISL checks LGLCOM for error codes. If an error was reported by LGRCRT, it is reported to the user through the Pascal interface routines.

NOTE

If a disk image of a system is being modified, the log files are never recreated, regardless of the value passed in the PARMS list.

6.5.10 Terminate SCU Session.

SCI Command: QSCU

Opcode: 12

Overlay: SCUINIT

PARMS:

The fourth element on the PARMS list is an abort indicator (YES/NO).

Normal termination of an SCU session is handled by the procedure CUQUIT. This procedure checks the abort parameter. If a value equal to NO is specified, CUQUIT issues Segment Manager SVCs to force write the root segments.

Although the request is meaningless, no error is generated when termination processing is requested with a value of YES passed as the abort parameter, and the running system is being modified.

6.5.11 Modify System Parameters.

SCI Command: MSP

This function of SCU is done in four stages. This design was chosen for the following reasons:

* Some terminals have as few as 12 display lines. This hardware limits the number of field prompts, so that all information cannot be displayed on one screen.

* Fewer synonyms are required for passing information between SCI and SCU than if all parameters are handled at once.

* The parameters that may be modified fit rather naturally into three categories.

The PARMS list used in bidding SCU for modifying system parameters consists of only the first four parameters.

Each stage is performed by a pair of Pascal procedures: CURSP1 and CUMSP1, CURSP2 and CUMSP2, CURSP3 and CUMSP3, and CUMSP4 and CURSP4. Procedures with an R in the name set synonyms to the current values in the system tables. Procedures with an M in the name store synonym values in the system tables. All procedures are in the module CUMSP.

Each of the CUMSP procedures depends heavily on two subroutines, RETRIEVE and RETURN. These two subroutines are used to map synonym values to binary values and map binary values to synonym values, respectively. Since values are of various types (for example, character, integer, or list), RETRIEVE and RETURN are called with a parameter to indicate the kind of value being passed or requested. These calls are made with a knowledge of system table formats. Should those formats change, changes to code that calls RETRIEVE or RETURN will be required.

6.5.11.1 Stage One.

In the first stage, values of miscellaneous system variables are returned (opcode 13), and modified (opcode 16).

Overlay: SCUMSP

Synonyms:

Synonym	Contents	Field Prompt Name
$CU$$DSPFG1	Statistic to display on left side of front panel	FRONT PANEL DISPLAY-LEFT
$CU$$DSPFG2	Statistic to display on right side of front panel	FRONT PANEL DISPLAY-RIGHT
$CU$$UNTSLC	Number of clock ticks in a time slice	CLOCK TICKS/SLICE
$CU$$ENDLMT	Number of STUs a task is allowed for end-action	END ACTION LIMIT(STU'S)
$CU$$MEMTIC	Number of ticks between parity checks	MEMORY ERROR SAMPLE RATE
$CU$$JCA	Number of bytes in a medium JCA	MEDIUM JCA SIZE

6.5.11.2 Stage Two.

In stage two, parameters used in scheduling are returned (opcode 14) and modified (opcode 17).

Overlay: SCUMSP

Synonyms:

Synonym	Contents	Field Prompt Name
$CU$$INTPRI	Initial run time priorities	INITIAL PRIORITY VALUES
$CU$$JPRMOD	Weighting factors for job priority on run time priority	WEIGHT OF JOB PRIORITY
$CU$$DYNMOD	How much to vary run-time priority for I/O bound tasks	DYNAMIC PRIORITY RANGE
$CU$$AGEIND	Whether to age run-time priorities	AGING ON PRIORITY
$CU$$IOINDX	Average time a task suspends	TICS BETWEEN SUSPENDS

6.5.11.3 Stage Three.

In stage three, swapping parameters are returned (opcode 15) and modified (opcode 18).

Overlay: SCUMSP

Synonyms:

Synonym	Contents	Field Prompt Name
$CU$$CLMXBF	Maximum number of buffers or segments to be cached	CACHABLE BUFFERS
$CU$$CLMXPS	Maximum number of program segments cached	CACHABLE PROGRAM SEGMENTS
$CU$$TLSPND	Minimum number of STUs in suspension until task is swapped	MINIMUM SUSPENSION TIME
$CU$$TLEXEC	Minimum number of STUs execution until task is swapped	MINIMUM EXECUTION TIME
$CU$$TOLS24	Whether to swap queue servers	STATE >24 IMMEDIATE ROLL?
$CU$$LDRTDY	Number of STUs to delay task loader	LOADER TIME DELAY(STU'S)
$CU$$JCARES	Minimum JCA free space prior to expansion	JCA EXPANSION BOUNDARY

6.5.11.4 Stage Four (more miscellaneous values).

Overlay: SCUMSP

Synonyms:

Synonym	Contents	Field Prompt Name
$CU$$JOBLMT	Maximum number of active foreground jobs	FOREGROUND JOB LIMIT
$CU$$JOBBLM	Maximum number of active background jobs	BATCH JOB LIMIT
$CU$$MEMSIZ	Size (in beets) of physical memory	PHYSICAL MEMORY SIZE
$CU$$SITENM	Site name (e.g. AUSTIN)	SITE NAME

6.5.12 Modify Device State.

SCI Command: MDS

Opcode: 19

Overlay: SCUDEL

PARMS:

PARM No.	Definition
4	Device name
5	New device state
6	Does device accept eight-bit characters?
7	Read after write error check?
8	Bit map read after write error check?

The modify device state request is processed in procedure CUMDS. It applies the specified changes directly to the PDT of the device being modified.

The read after write error check can only be applied to the running system. This rule is enforced in the code. The feature is not allowed on DS31 drives, because the DS31 controller does not comprehend the transfer inhibit bit. SCU issues a store registers SVC to the device to determine whether it is a DS31. Because this test is always done, the read after write error check cannot be activated for a drive that is off-line.

6.6 MODIFYING SCU

In general, the use of Pascal to implement SCU facilitates sustaining or even adding capabilities. SCU does not use the Pascal I/O package. The only I/O that SCU explicitly performs is through synonyms and SCI prompts or through calling S$ routines to write lists to the output file. The fact that the main driver is for all practical purposes a single CASE statement based on the opcode makes adding new capabilities to SCU straightforward; simply add a new case and define a new procedure to process it.

Note that SCU does some implicit I/O by using updatable program file segments. During processing of the request to terminate an SCU session, the root segments are force written to their home file unless the session is aborted by the user.

6.6.1 Coding Conventions.

The SCU code uses copy files for constants and types that may need to change. Constants are used extensively, in an effort to localize potential SCU modifications due to data structure or command procedure changes. Constants are defined for the following:

* Error message codes (message numbers)
* Overlay IDs
* SCI PARMS list positions
* Synonym name mapping values (except for procedures CURISL and CUISL)
* SCU opcodes

The SCU code follows the DNOS naming conventions as described in the DNOS System Design Document, with regard to module names and source file names. The exception is that subroutines (procedures) defined and used locally in a single major subroutine are given names that have meaning. For example, the data base initialization routine, CUDATA, contains subroutines ADD_SINGLE, ADD_MULTIPLE, and so on; CUMDS, which modifies device state, contains a function IN_USE.

The SCI command procedures that invoke SCU are two-phased; that is, they return defaults and then apply user-specified changes. Most command processors consist of two procedures. The procedure that applies the command is named CUxxx, where xxx is, in general, the command procedure name. The Pascal procedure that returns defaults is named CURxxx, with the R indicating return.

6.7 INTERNATIONALIZATION

The output produced by SCU consists of numbers, device names, and report text (for example, LDC headings, device attributes). The report text is all contained in the only SCU assembly language modules. The text is in the form of Pascal strings, with each string in a separate CSEG. In addition to report text, CUCOM contains the text for several necessary constants, such as YES, NO, TRUE, FALSE, and the device states (online, offline, diagnostic, spooler). Whenever code within SCU must determine whether the answer to a prompt is YES/NO, TRUE/FALSE, etc., it compares the prompt answer to an appropriate string in CUCOM; thus, there is no imbedded text in any SCU module except CUCOM.

The country code is handled as an integer by the program. All mapping of integer to country name is done within the command procedure.

6.8 COMPANION COMMAND PROCEDURES

The user interface to SCU is SCI.

Table 6-4 summarizes SCU functions, grouped by command procedures as shipped by Texas Instruments Incorporated. The commands that are available outside an initiated session are noted. System modifications made outside an initiated session cannot be made permanent in the same way as modifications made during an SCU session. Modifications are made permanent by updating the home program file. Updating is done as part of session termination. Some SCU commands make changes to system data structures that become effective only after an IPL (for example, adding a new type of device, which requires a new DSR for that device; changing system table sizes).

System modifications made outside an active session are applied to the running system.

Table 6-4 SCU Commands

XSCU	- Execute SCU
LDC	- List Device Configuration (Note 1)
MDC	- Modify Device Configuration
SCC	- Show Country Code (Note 1)
MCC	- Modify Country Code
MST	- Modify System Table size (Note 2)
ISL	- Initialize System Log (Note 1)
MDS	- Modify Device State (Note 1)
MSP	- Modify Scheduler/Swap Parameters (Note 1)
QSCU	- Terminate an SCU Session

Note 1 - Command is available during uninitiated sessions.

Note 2 - Command is available in limited form during uninitiated sessions.

6.8.1 Command Procedure Design.

All command procedures to modify the system are written to flow as follows:

* RBID SCU with the opcode specifying that the utility is to return the current values of parameters associated with the item to be modified.

* Prompt the user for new values, using the current values as defaults.

* RBID SCU with the opcode specifying that the utility is to modify the image in memory. The values supplied by the user are passed to SCU in either the PARMS list or in $CU synonyms.

Nothing in the code requires that SCU be called first to return values. The first of the above steps could be omitted.

SCU does not validate the values passed on the PARMS list. The command procedures, as shipped by Texas Instruments Incorporated, ensure that all data entered by the user is of a proper type and has an acceptable value (for example, range or element of a list). Changing the field prompt declarations in SCI command procedures that invoke SCU could allow a user to make disastrous changes to the operating system being modified.

6.8.2 MDC Command Procedure Package.

MDC is the top level of a three-tiered command procedure structure. This hierarchy minimizes the number of command procedures required to specify all possible modifications (add, delete, or change) to a total of twelve devices. A discussion of the structure follows, along with an example session to show how MDC steps the user through the process of changing the characteristics of a device.

All other SCU command procedures are the more conventional single-level type and are not discussed in this document.

The MDC hierarchy is shown in Figure 6-8.

```
                          MDC
                           |
            +--------------+--------------+
            |              |              |
          MDC$A          MDC$C          MDC$D
       Add Device    Change Device    Delete Device
            |              |
            +---------+----+
                      |
   +----+---+----+----+----+----+----+----+--+---+----+----+
   |    |   |    |    |    |    |     |   |  |   |    |    |
MDC$010 |   |  MDC$015|  MDC$03 |  MDC$08 |  | MDC$0B | MDC$TYP
 Card   |   |  Virtual|   ASR   | Magnetic|  | Serial | Set the
Reader  |   | Terminal|  Device |Tape Unit|  |  Line  | Synonym
        |   |   Base  |         |         |  | Printer|  $TYP
        |   |         |         |         |  |        |
        |MDC$012    MDC$02    MDC$07      |MDC$0A    MDC$0C
        |931 VDT     KSR       Disk       | 911      Parallel
        |           Device    Drive       | VDT        Line
        |                                 |          Printer
     MDC$011                            MDC$09
     940 VDT                             TPD
```

Figure 6-8 MDC Command Procedure Structure

MDC prompts the user for the kind of change to be made (add, change, or delete), and invokes the appropriate procedure at level two.

Level two procedures prompt the user for device name and/or device type. At this level, all information necessary to delete a device is known. MDC$D RBIDs SCU and does not invoke any procedures at level three.

MDC$A and MDC$C invoke the appropriate third-level procedure. The name of the third-level command procedure is constructed by appending the value of the synonym $TYP to the character string MDC$. $TYP represents the device type as defined in the LDT template.

Level three procedures prompt the user for specific information needed to define a device of the particular type. SCU is then RBID with the proper opcode and PARMS list.

The following example shows the technique used in writing the MDC command procedure set. Explanatory comments have been added, and the procedures are not exact replicas of those shipped with DNOS.

Assume that you have invoked MDC. The following command procedure is executed:

```
          MDC(MODIFY DEVICE CONFIGURATION)=6,                  1
          DATA DISK/VOLUME=*ACNM("@$XSGU$D")                   2
          .SYN $XSGU$D="@&DATA DISK/VOLUME",                   3
          $MDC$DD="@@$XSGU$D .S$OSLINK.S$SGU$"                 4
          .LOOP                                                5
          .PROMPT (MODIFY DEVICE CONFIGURATION),               6
          COMMAND(CHANGE,ADD,DELETE)=ELEMENT(C=C,              7
          A=A,D=D)(C)                                          8
          MDC$&COMMAND                                         9
          .UNTIL @$$CC,NE,0                                    7
          .UNTIL @$$MO,EQ,0                                    11
          .REPEAT                                              12
```

Lines 5 through 12 are a loop. Line 9 is the construction of the second-level command procedure name. Line 10 provides for exiting the loop if an irrecoverable error occurs and line 11 prevents looping in batch mode.

Assume you responded ADD to the COMMAND field prompt. The command procedure MDC$A is invoked.

```
MDC$A (ADD DEVICE)=6,                                                  1
DEVICE TYPE = ELEMENT(VDT=VDT,911=VDT,DISK=DISK,                       2
MAG TAPE=MAG TAPE,VIRTUAL TERMINALS=VIRTUAL TERMINALS,                 3
ASR=ASR,KSR=KSR,CARD READER=CARD READER,                               4
LINE PRINTER=LINE PRINTER)                                             5
.RBID TASK=02E, UTILITY, PARMS=(500,800,8,03000)                       6
.IF @$$CC, EQ, 0                                                       7
.SYN $SCU$BTA = @$CU4                                                  8
.SYN $TITLE="ADD &DEVICE TYPE",$OP=4,                                  9
$CU1="",$CU2="",$CU3="",$CU4="",$CU5="",$CU6="",$CU7="",               10
$CU8="",$CU9="",$CU10="",$CU11="",$CU12="",$CU13="",                   11
$CU14="",$CU15="",$CU$16="",$CU17=""                                   12
MDC$TYP DEVICE TYPE ="&DEVICE TYPE"                                    13
.IF @$TYP, IS, ELEMENT(0A,03,02,015)                                   14
.SYN $CU15=">@$TYP",$CU16=">E000"                                      15
.ENDIF                                                                 16
.SYN $MDC$AD="Y"                                                       17
MDC$@$TYP                                                              18
.ENDIF                                                                 19
.SYN $TYP="",$TITLE="",$PROC="",$OP="",$MDC$AD="",                     20
$CU1="",$CU2="",$CU3="",$CU4="",$CU5="",$CU6="",$CU7="",               21
$CU8="",$CU9="",$CU10="",$CU11="",$CU12="",$CU13="",                   22
$CU14="",$CU15="",$CU16="",$CU17="",$SCU$BTA="",$SCU$INT=""            23
```

Line 18 is the construction of the third-level command procedure name.

Assume you responded DISK to the DEVICE TYPE field prompt. Since this is a disk, TYP has the value 07 (the 7 was set by SCU the first time it was invoked, and the 0 was added in the command procedure). Command procedure MDC$07 is invoked.

```
MDC$07 (@$TITLE)=6,
TILINE ADDRESS = RANGE(0F800,0FBF0)("@$CU2"),
DRIVES = RANGE(1,4)("@$CU6"),
DEFAULT RECORD SIZE = INT("@$CU7"),
INTERRUPT = RANGE(3,15)("@$CU3"),
EXPANSION CHASSIS = *RANGE(1,7)("@$CU4"),
EXPANSION POSITION = *RANGE(0,23)("@$CU5")
*BID TASK SCU
.RBID TASK=02E,UTILITY,PARMS=(500,800,@$OP,@$CU1,&TILINE,
&INTERRUPT,&EXPCHAS,&EXPPOS,@$TYP,
&DRIVES,,,,0,NO,
,,,&DEFAULT,,,,,,,,,,,,,0)
```

This command procedure bids SCU with the appropriate PARMS list to add a disk in the system currently being configured.

SECTION 7

OPERATOR INTERFACE

7.1 OVERVIEW

The operator interface subsystem provides a mechanism by which information and/or requests are communicated to a user who is performing operator functions, and to any other users who ask to see the messages.

The operator interface is designed to enable any task in any job to pass a request to the system operator when the task requires intervention (for example, when devices require attention). The request may or may not require a response from an operator. If it does, the subsystem allows the task to specify time-out information. The requesting task is notified when an operator responds to the request, or when the specified time has expired, whichever occurs first.

Operator functions may be either centralized to one user, who is called the system operator, or distributed among all users who have requested to have the text of operator requests displayed at their terminals.

The basic design of the operator interface subsystem is passive. An operator must initiate the transaction of responding to a specific request. The subsystem maintains a list of pending requests, but takes no initiative to get operator response.

While the operator interface subsystem does service a series of requests, it is not a queue server in the same sense as the Job Manager and the Name Manager. The operator interface subsystem does not accept input from a batch job. It is written in Pascal and requires the following support:

* MAILBOX services

* Interprocess communication

* Initiate Event and Wait for Event SVCs

* Access to a system disk for a temporary file

7.2 STRUCTURE

The operator interface subsystem is implemented in two tasks -- the system operator task, OPERATOR, and the operator interface task, XOI.

7.2.1 System Operator Task.

The system operator task, OPERATOR, is a nonreplicatable, nonprivileged system task. OPERATOR is bid in the system job by the system restart task, and, once through the initialization phase, never terminates.

The functions of the OPERATOR task are as follows:

* As the owner and master of S$OPER, processes all writes and reads issued to the channel

* Maintains the operator request table, a list of all pending operator requests

* Maintains the user ID table (UIDT), a list of all users who have requested that operator requests be displayed at their terminals

* Maintains the pointer in the system common area NFPTR to indicate which user, if any, is currently the system operator

7.2.2 Operator Interface Task.

The operator interface task, XOI, is bid in the user's job by SCI. XOI is replicatable and nonprivileged.

The function of XOI is to interface between a user and the OPERATOR task. This consists of writing messages on the channel and processing the associated reply buffers.

For the system operator whose terminal is dedicated to the system operator function, XOI displays incoming requests and provides special processing of two function keys for responding to and killing specific requests.

7.3 COMMUNICATION BETWEEN TASKS

OPERATOR is the owner of .S$OPER, a global master/slave interprocess communication (IPC) channel. All communication between requesting tasks and the system operator task is done on .S$OPER.

As owner of the channel, OPERATOR is responsible for processing Open and Close SVCs on .S$OPER. OPERATOR does not open the channel to any access privilege other than shared.

All messages on .S$OPER must be sent with a write operation using the reply option. Any message received by OPERATOR that is not so written is not processed; an error code that indicates an invalid opcode is put into the SVC block and the write to the channel is terminated.

Only the XOI task of the system operator is allowed to issue a read to .S$OPER.

The format of messages written to .S$OPER is covered in the paragraph on detailed design of the operator interface task, XOI, which has responsibility for formatting the appropriate message. The format of the reply buffer is in the detailed design of the system operator task, OPERATOR, which has responsibility for formatting the reply.

7.4 GENERAL DESIGN CONCEPTS

The following paragraphs contain information about terms used in this document, the rules of operator privilege, and the format of messages displayed by the operator interface subsystem.

7.4.1 Definitions.

The following terms are used throughout this document. They are not used in comments that appear in the source code, but are defined in this document to clarify concepts in the operator interface subsystem.

* Operator - A user who has executed the ROM command, without a subsequent KOM, or who is the system operator

* System Operator - A user who has been designated the system operator by successfully executing the XOI command (without a subsequent QOI)

* (Operator) Request - An entry in the operator request table

* (Operator) Response - The data produced by operator action with regard to a request

* Requester - The user task with which an operator request is associated. The requester may be XOI or another task. For example, the Spooler generates requests in behalf of the user who invoked the Spooler. In this case, the user task that invoked the Spooler is the requester.

* Message - An OPERATOR opcode and the supporting data to obtain services from OPERATOR

* Reply - A buffer of information that is the result of services provided by OPERATOR

NOTE

The system task in the operator interface subsystem is installed with the name OPERATOR. In the code, it is often called SOT (for system operator task). In this document, the system task is called OPERATOR. This is done to avoid confusion between a reference to the system task and a reference to the XOI task associated with the user who is designated the system operator (the system operator's XOI task).

7.4.2 Operator Privilege.

The operator interface subsystem enforces the following restrictions concerning operator privilege:

* If a user is designated the system operator, only that user is allowed to respond to operator requests, although many other users may still be viewing the text of the requests.

* If no user is designated the system operator, several users may be eligible to respond to operator requests. When the user's ID is put on the list of those to whom requests are relayed, the user becomes eligible to respond to any request which would be relayed to him.

OPERATOR places one further restriction on the system operator, in that only one session (terminal) is considered the system operator. This is enforced by requiring that not only the user ID and job ID of a requester be the same as the system operator, but that the task ID in the TSB must also match. Thus, even though DNOS allows a user ID to be concurrently logged on at several terminals, only the session associated with the terminal where XOI was successfully executed is recognized by OPERATOR as the system operator. Other subsystems may allow the system operator to use multiple terminals, but the operator interface subsystem does not.

Other subsystems may grant to the system operator privileges that are not generally available. For example, the Spooler subsystem allows the system operator to kill any request in any output queue.

7.4.3 Transactions.

The construct of a transaction is useful in understanding the flow of the operator interface subsystem. A transaction is the process by which a piece of business is accomplished.

The operator interface subsystem conducts the following kinds of transactions:

* A transaction that consists of one message/reply pair. The reply may be written immediately or after some interval, but when the reply is written, the transaction is complete.

* A transaction that consists of more than one message/reply pair. A message is written to .S$OPER that initiates processing for a piece of business that cannot be completed before the reply is written. At least one more message/reply pair is required to complete the piece of business.

* A read message from the XOI task of the system operator. This transaction differs from the first two in that it is a read to .S$OPER, rather than a write.

The flow of the first transaction type is simple. This category includes all general requests and all SCI commands except KOR and ROR. As an example, when a user enters the ROM command, the XOI task is bid by SCI in the user's job. XOI writes a formatted message on .S$OPER. OPERATOR adds the user's ID to the list (UIDT) and replies to the user's XOI task, completing the write on .S$OPER. The XOI task in the user's job then terminates, unless the user is designated the system operator.

With the second transaction type, XOI writes the first message of the transaction, processes the reply buffer, then writes a second message and processes that reply. XOI then terminates (unless the user is designated as the system operator) because the transaction is complete. OPERATOR processes both messages and writes both replies. This category includes the SCI commands KOR and ROR, commands that prompt the user for information used in formatting the second message to OPERATOR.

The third transaction type occurs only when a system operator has dedicated the terminal to system operator functions. The timing of completing the read depends upon whether there is a request that has not yet been displayed to the system operator. If there is such a request, the read is processed immediately. If not, the read to .S$OPER is left open until such a message does exist. There is no message buffer for this transaction.

7.4.4 Format of Displayed Requests.

Operator requests are always displayed in the following format:

```
          OR xxxxxf FROM user AT hh:mm-general text
```

where:

xxxxx	is the request ID or blank.
f	is the response mark -- an asterisk if a response is required and a blank otherwise.
user	is the user ID associated with the request.
hh:mm	is the time the request was received by OPERATOR, in hours and minutes.
general text	is the general text of the request.

When a request is displayed through MAILBOX, the entire message just shown is sent as the text of a MAILBOX message, with one exception. If the user to whom the request is being displayed is not allowed to respond to the request (that is, if another user is the system operator), then the request ID field, xxxxx, is blank. When the message is displayed at the user's terminal, MAILBOX headers are appended to the front of the message.

7.5 SYSTEM OPERATOR TASK

OPERATOR performs the following functions:

* Processes general operator requests. Places a request in the operator request table (ORT)

* Initiates relay operator messages (ROM). Creates an entry in the UIDT for this user ID

* Terminates relay operator messages (KOM). Deletes the user's ID from the UIDT

* Creates a file containing a list of (pending) operator requests

* Designates a user as the system operator

* Terminates a user as the system operator

* Processes the transaction whereby an operator responds to or kills a specified pending operator request

OPERATOR is bid by the restart task during IPL.

7.5.1 Data Structures and Files.

OPERATOR maintains one item in the operating system common area, three major data structures for its own use, and task local variables to control flow through the code.

7.5.1.1 OPERATOR Local Variables.

The OPERATOR local variables are as follows:

* REPLY - Flag to indicate whether or not to write a reply

* NXTREQ - The request ID for the next request placed in the operator request table

7.5.1.2 System Common Area.

OPERATOR maintains the pointer SOPJSB in NFPTR, which points to the JSB of the system operator's job. SOPJSB is zero when no user is acting as the system operator.

7.5.1.3 System Operator Information.

OISOPR is a data structure maintained by OPERATOR. It contains the following information concerning the current system operator:

* OPRSTT. Operator state:

  - XOI. Requests are to be sent to the system operator when a read is issued by XOI to .S$OPER. This is the state when the user is first designated the system operator. If the system operator enters ROM mode and then leaves ROM mode, XOI state is reinstated.

  - ROM. The system operator's terminal is not dedicated to XOI; the text of an incoming request is sent through MAILBOX.

* OPRJID. Pointer to the job ID associated with the system operator. A value of zero implies that no user is designated the system operator.

* OPRTSB. Pointer to the TSB of the system operator's XOI task.

* OPRRDP. Pointer to the reply buffer associated with a pending read on .S$OPER. A value of zero means that no read is pending.

* OPRRPT. Pointer to the ORT entry to which the system operator is currently responding. A value of zero indicates that the system operator is not currently responding to a request.

7.5.1.4 Operator Request Table (ORT).

The ORT is maintained by OPERATOR. The structure is a circularly linked list of Pascal records in the OPERATOR task area. The header record in memory points forward to the next record and backward to the final record. In addition to the linking information for the entire ORT, entries that specify a time-out are circularly linked to form the time-out list.

ORT entries remain indefinitely until they are removed for one of the following reasons:

* An operator responds to the request

* An operator kills the request

* The request exceeds the time-out limit

The ORT contains one record for each pending request. The following information is in each ORT record:

* Pointer to the Master Read/Write buffer (MRB) that contains the information in the following list. The format of the message written in this buffer is covered in the paragraph on detailed design of XOI.

  - Information needed to do a Master Write to .S$OPER

  - General text

  - Prompt information (maximum of two prompt/initial value pairs)

* Request ID (1 through 65,535). These decimal numbers are assigned sequentially as requests are placed in ORT.

* User ID of requester

* Flags:

  - Response required?

  - Written to the system operator yet?

* Time of request

* Time-out value

* Response state. A pointer to the UIDT entry for the user who is currently responding to this operator request and to minus one when the system operator is responding to the request. This pointer is set to zero when no user is responding to the request.

7.5.1.5 User ID Table (UIDT).

The UIDT is a list of all active users who wish to receive and possibly respond to operator messages. This list is maintained by OPERATOR. The structure is a circularly linked list of Pascal records in the OPERATOR task area.

Each entry contains the following information:

* User ID

* Job ID

* Pointer to the JSB of the job with which this user ID is associated

* Station number with which the user is associated

* Flags:

  - UIDALL. Whether the user wants all operator messages (T) or only those originating from or directed to this user ID (F)

  - UIDOPR. Whether this user is the system operator

* Pointer to the operator request to which this user is currently responding

7.5.2 Initialization.

System operator task initialization is done by the procedure OISINT, in the module of the same name. The operator channel .S$OPER is deleted and created. A LUNO is assigned to the channel and it is opened.

Internal buffers, pointers and variables are initialized. The pathname .S$OPMSxx is stored in the variable LOMNAM for use in building the pathname of the file where the formatted list of operator requests is temporarily written. (When a user requests a list of pending operator requests, xx is replaced with the user's station ID). The ORT and UIDT linked lists are initialized with only one record in each. The one record in ORT is linked to itself with respect to both the pointers for the entire list and pointers for the time-out list. OISOPR is initialized to reflect that there is no system operator, and that no read is pending on .S$OPER.

7.5.3 Major Loop/Routines.

OISTSK is the name of the Pascal program that executes in the OPERATOR task. The program is in module OISTSK. The major loop consists of the following logic:

```
LOOP1:DO forever;
       IF the time-out list is empty
          THEN
             Issue a master read with suspend to .S$OPER;
             Process the I/O from .S$OPER;
          ELSE
             LOOP2: For all requests on time-out list;
                IF Request pending longer than specified time-out
                   THEN Write reply with time-out code;
                        Remove request from ORT;
             END LOOP2;
             Issue a master read without suspend;
             IF I/O was returned from .S$OPER
                THEN Process the I/O;
                ELSE Suspend for five seconds;
END LOOP1;
```

7.5.4 Error Processing.

Errors are reported to requesters in the error code field of the reply buffer for S$OPER. Error code values returned by OPERATOR are defined in the Pascal template OISCONS.

Errors resulting from SVCs issued by OPERATOR are processed in the routine OISERR. With one exception, all such errors are written to the system log. The error caused by writing a reply to a task that has terminated is ignored.

7.5.5 Termination.

OPERATOR is designed never to terminate. The only error that is irrecoverable occurs in the initialization phase. If OPERATOR is unable to create, to assign a LUNO to, or to open the IPC channel .S$OPER, the error is written to the system log and the task is terminated through Pascal end-action.

7.5.6 Detailed Design.

One major function of each OPERATOR processing routine is to prepare the reply buffer for a particular message. The contents of the reply buffer vary with the opcode in the message and the results of the processing done by OPERATOR. The details of the reply buffer format are discussed with each of the processors in the following paragraphs.

7.5.6.1 OISXOI.

OISXOI processes the message for designating a user as the system operator.

If no system operator already exists, OISKOM is called to take the user out of the UIDT (the user may have previously entered the ROM command), the requester's JSB pointer is stored in NFPTR.SOPJSB, and the OISOPR data structure is set up with the following characteristics:

* System operator not responding to any operator request
* No read pending on S$OPER
* System operator receiving messages in XOI mode

If a system operator has already been designated, and is not the requester, an error code indicating invalid request is put into the reply buffer. The requesting user is notified by the XOI task that he or she has not been designated the system operator. If this is a request from the system operator, OISKOM is called to remove the operator's user ID from UIDT and set the operator mode to XOI.

The reply buffer for this message is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=5)
1/1	Error code

7.5.6.2 OISQOI.

OISQOI processes the message to terminate the designation of the user as system operator.

Before any action is taken on the message, the job ID and TSB of the requester are compared to the data in OISOPR. Unless both items match, the request is denied, an error code is written in the reply buffer and processing in OISQOI terminates. The XOI task informs the user of errors.

The next consideration is whether the system operator is in ROM mode or XOI mode. If ROM mode, OISKOM is called to take the operator out of UIDT. In XOI mode, if a read is pending on .S$OPER, it is killed with an error code in the reply buffer to so indicate. The pointer to the read reply buffer is cleared, resetting the read pending flag.

Next, OISQOI clears two pointers to the system operator -- its own data structure element OPRJID and SOPJSB in the system common area NFPTR.

If the operator was in the process of responding to a request, the response state of that ORT entry is cleared.

Now that the system operator has been effectively relieved of that function, each entry in the ORT must be updated as follows:

* Mark each entry as not having been sent to the system operator.

* Call OISMBX to send each message to the appropriate users in the UIDT. This time, the request ID field is nonblank so that any user who so desires may respond to the request. Note that the user who was formerly the system operator does not receive these messages, as that ID has just been deleted from the table by OISKOM.

An error code of zero is placed in the reply buffer and OISQOI processing ends.

The reply buffer format is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=6)
1/1	Error code

7.5.6.3 OISRD.

Procedure OISRD processes read messages on .S$OPER. A nonzero error code is placed into the reply buffer in the following cases:

* The requester is not the system operator (a task that is not the system operator's XOI issues the read).

* The requester is the system operator, but is not in XOI mode.

* A read is already pending from the system operator.

Otherwise, the ORT is searched for the first entry that has not yet been sent to the operator. Since entries are added to ORT as they are received, the first entry found is the oldest pending request not yet displayed to the system operator. When an unsent request is found, the text is formatted into the reply buffer and the ORT entry is marked as having been sent to the operator.

If no unsent entry is found, the OPERATOR variable REPLY is set to a value of false. The MRB is saved and a pointer to it stored in OPRRDP. A reply is written on S$OPER to complete the read only when the value of REPLY is true.

The format of the reply for the read message is as follows:

Offset/ Byte Length	Description
0/1	OPERATOR opcode (=0)
1/1	Error code (zero except when read not honored)
2/2	Request ID
4/8	User ID
12/5	Time of request
17/1	Response mark: A blank means no response is required. An asterisk means a response is required.
18/1	Length of general text
20/?	General text (maximum of 223 bytes)

7.5.6.4 OISLOM.

OISLOM processes the request to list operator requests, either all of them or a subset consisting of those associated with the requester's user ID.

An error code indicating that the request is invalid is set in the reply buffer if the user is neither the system operator nor in the UIDT. Another error code is set in the reply buffer if there are no requests (entries in the ORT) to be displayed.

The next section of OISLOM is a loop on entries in the ORT. The request is written to the temporary file .S$OPMSxx (the complete pathname is stored in the variable LOMNAM), if one of the following is true:

* The requester is the system operator.

* The requester specified that all requests be relayed.

* The requester's user ID is the same as the user ID in the ORT entry.

The variable MSGSENT is maintained to determine whether a request is found in ORT that satisfies one of these conditions and is actually written to the file.

For each request to be listed, procedure OISFMS is called to format the general text. If the requester is the system operator or if there currently is no system operator, the request ID is filled in. Otherwise, blanks are written into that field in the formatted text.

The message is then written to the temporary file, in 80-character lines, breaking on blanks (or in column 60 if no blank occurs past that point).

After all entries in the ORT have been processed, the variable MSGSENT is tested, and if it is false, the error code indicating no messages is set in the reply buffer.

The format of the reply buffer is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=4)
1/1	Error code

7.5.6.5 OISROM.

OISROM processes the message to add a user ID to the UIDT.

If the user ID is already in UIDT, the entry is updated to reflect the current specification for whether the user is to receive all operator requests or only those associated with the user ID, and an error code indicating no errors is returned to the requester in the reply buffer. Otherwise, a new UIDT entry is created and chained into the UIDT list, and all appropriate requests in the ORT are sent to the user through MAILBOX.

The new UIDT entry has the following characteristics:

* This user is currently not responding to a request so the pointer is zero.

* UIDOPR is either true or false, depending on whether this user is the system operator.

* UIDALL is set according to the following rules:

  - True if this is the system operator or if the message specifies that the user wishes to see all operator requests

  - False otherwise

NOTE

> Even if the system operator enters a request to be shown only the user's own messages, the UIDT entry is built in such a way that the system operator receives them all. The system operator is not allowed to change this item in UIDT with subsequent ROM commands.

If the requester is the system operator, the data structure OISOPR is updated to show that the system operator is now in ROM mode. If a read is pending on S$OPER, it is terminated with an error code indicating an abort, and the read pending pointer is cleared.

For each message that has not previously been sent to this operator, OISMBX is called. If the user is the system operator, the logic employed does not send messages through MAILBOX that have already been sent on .S$OPER to the operator in XOI mode.

The reply buffer format is:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=2)
1/1	Error code

7.5.6.6 OISKOM.

OISKOM processes a message to remove a user's ID from the UIDT (and therefore, discontinue relaying operator messages to that user through MAILBOX).

In order to remove any ambiguity concerning which ID is to be removed from the table, several precautions are taken. Not only must the user ID match, but also the job ID that placed the user in UIDT must match the ID of the job in which the message to remove the ID was generated. If such a match is not found, an error code is set to indicate invalid KOM request.

If the request is valid, the previous and next UIDT entries are altered to unchain the record for the user ID being deleted. If the UIDT entry shows that the user was in the process of responding to an operator request, the appropriate ORT entry is cleared to show that the request is no longer in response state.

If the user being removed from UIDT is the system operator, the status of the system operator is changed to XOI mode. The memory used for the deleted entry is released and an error code of zero is set.

The format of the reply buffer is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=3)
1/1	Error code

7.5.6.7 OISGRQ.

Procedure OISGRQ processes a message that contains an operator request to be added to the ORT. The message is examined for consistency of data. OISGRQ generates an error code when any of the following circumstances is encountered:

* Flags are set that do not apply to a general request. Even though the flags would not cause erroneous processing of this particular request, their being improperly set casts suspicion on the remaining data.

* A response is not required, but the number of prompts is nonzero.

* Too many prompts are specified (the maximum number, MXPRCT, is declared in OISCONS).

* The length of the general text is either zero or greater than the maximum number of characters allowed. (Again, the maximum length of the general text, MXTXTL, is declared in OISCONS).

* The length of a prompt is either zero or greater than MXPRTL, the maximum prompt length, which is declared in OISCONS.

If the flag which indicates that the job ID is specified in the message buffer is set, OISGRQ runs the operating system JSB chain in search of an entry that has the specified job ID. If the job ID is not found, an error is set and processing in OISGRQ ends.

If the data is valid according to all these tests, a new ORT entry is created with the following characteristics:

* It is not in response state (no operator currently responding to this request).

* The value of NXTREQ is assigned as the request ID.

* If time-out is specified, the record is chained into the time-out list. Pointers in this record, the previous record, and the next record are updated to place the record in the list.

* The entry is marked not yet sent to the system operator.

* Each ORT entry contains forward and backward chaining. Pointers in this record, the previous record and the next record must be updated.

NXTREQ is incremented by one and the general text of the request is written to the system log. (For more details, see the previous paragraph that describes the format of displayed requests).

The next part of OISGRQ is concerned with sending the request to the system operator and/or any interested user. The variable REPLY is set to the complement of the flag that indicates whether a response is required. The variable REPLY determines whether or not a reply is written to the message on .S$OPER. If the request does not require an operator response, REPLY is given a value of YES so that the write on .S$OPER is completed. If the request does require a response, REPLY is given a value of NO. The requester remains suspended until the operator responds or kills the request or the request exceeds the time-out limit. The write on .S$OPER is not completed immediately.

Procedure OISMBX is called to send the message, through MAILBOX, to each user whose ID is in UIDT. If the system operator XOI has a read pending, the message is placed in a holding buffer and OISRD is called to format the reply buffer. The reply is written to .S$OPER to complete that read on the channel.

If the reply to the message just processed is to be written immediately (as reflected by the value of REPLY) the reply buffer is formatted for the general operator request, and written to .S$OPER to complete the write.

The format of the reply buffer is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=1)
1/1	Error code
2/1	Number of prompts (=0)

7.5.6.8 OISPOR.

OISPOR processes the initial (and possibly only) message in the transaction of an operator responding to a request. Information about the request is returned in the reply buffer.

An error code is returned in the reply buffer if the user is not allowed to respond to the specified operator request. Any of the following circumstances prevents the requester from being allowed to respond to the request:

* A system operator exists and the requester is not the system operator.

* The requester is not in the UIDT.
* The request is currently being serviced by another operator.
* The specified request ID is not in the ORT.

Once the requester has been cleared to respond to the request, the response required flag is checked. If no response is required, the ORT entry is unchained (on both the ORT list and the time-out list) and the error code in the reply buffer is set to indicate that no response is required for this operator request. This completes processing for requests to which no response is required.

If a response is required, this is the first of two messages required to complete the transaction. The text and the prompt information is put in the reply buffer. The response state is updated in the appropriate ORT entry, indicating that this operator request is now being serviced. OISOPR or the appropriate UIDT record is updated to show which ORT entry is being serviced by the operator. The format of the reply buffer is as follows:

Offset/ Byte Length	Description	
0/1	OPERATOR opcode (=7)	
1/1	Error code	
2/2	Request ID	

The remainder of the reply buffer is present only if the error code is zero.

Offset/ Byte Length	Description	
4/1	Filler for word boundary alignment	
5/1	Number of prompts	
6/1	Length of general text	
7/?	General text	
?/1	Length of first prompt (maximum of 28 bytes)	(Note 1)
?/?	Text of first prompt	(Note 1)
?/1	Length of default for first prompt (maximum of 50 characters)	(Note 1)
?/?	Text of default for first prompt	(Note 1)
?/1	Length of second prompt	(Note 2)
?/?	Text of second prompt	(Note 2)
?/1	Length of default for second prompt	(Note 2)
?/?	Text of default for second prompt	(Note 2)

Note 1 - Provided if number of prompts is one or two

Note 2 - Provided if number of prompts is two

7.5.6.9 OISROR.

OISROR processes the second message in the transaction of a user responding to an operator request.

An error code is written into the reply buffer if the specified request is not in the ORT, or the request is not in response state, or the request is in response state to a user other than the user ID associated with this message.

After checking all these conditions, OISPOR resets the response state of the request in the ORT. The appropriate data structure, either OISOPR or the UIDT, is updated to show that the operator/user is no longer occupied with responding to this request. Both the request and operator updates are done prior to processing the response data in the message, because even if the data is unacceptable, this two-part transaction must be repeated from the beginning.

If there are any irrelevant flags set, a nonzero error code is set and OISPOR processing ends.

The remainder of OISROR completes the transaction begun by processing in OISPOR. If there is no data to be considered (as in the case of the user hitting the Command key and aborting the response), the error code in the reply buffer is set to zero and no further processing is done. This represents the no response condition. The request is left in ORT and the only thing remaining to be done is to complete the XOI write on S$OPER. The reply buffer is already formatted.

If the response is negative (that is, if the operator killed the request rather than responding to it), OISRPL is called to do the following:

1. Format a reply that indicates the negative response from the operator

2. Send the reply to the task associated with the ORT entry

The negative response is processed here because common code is used to kill and respond to an operator request.

Otherwise, the response is positive unless one of the following errors is found in the data:

* Number of prompts returned does not match the number of prompts in the ORT entry.

* Data overflow is detected in value(s) returned for prompt(s).

If neither of these conditions is found, OISRPL is called to format and send a positive response to the task that initiated the operator request.

NOTE

> Only the positive and negative responses are generated in OISROR. The other response that may be sent to the task that initiated the request is time-out, which is generated in OISTSK. The reply buffer format is detailed in the discussion of OISRPL.

If a reply to the operator request was written, the master read buffer must be restored so that the proper reply is written to the XOI that supplied the operator response data.

The format of the reply buffer for the messages processed by OISROI is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=8)
1/1	Error code

7.5.6.10 OISRPL.

This is a service routine that sends a reply to the task that generated an operator request. It is called with two arguments; the first is a pointer to the ORT entry to which the reply applies and the second is the type response to be sent -- positive, negative, or time-out.

The disposition of the request is written to the system log, as follows:

```
**** OI - REPLY TO nn    :disposition
```

where:

nn	is the request ID.
disposition	is one of the following:

* TIMEOUT - Request timed out prior to operator response.
* REQUEST DENIED - Operator killed the request.
* REQUEST GRANTED - Request with zero prompts was granted.
* The prompt(s) and the response(s) of the operator if the request has one or two prompts.

The error code is set, depending on the response type. The error codes are defined in OISCONS.

The reply is written and the entry is removed from the ORT list and the time-out list.

If the reply is written because of a time-out, some additional processing must be done. If the request is in the response state, the operator who was servicing the request must be relieved of that burden. OISOPR or the UIDT entry is updated to reflect the fact that the request has timed out.

For the general operator request, the reply buffer format is as follows:

```
  Offset/Byte Length   Description
  ------------------   -----------
          0/1       Opcode (=1)
          1/1       Error code
The remainder of the reply buffer is present only if the
error code indicates a positive response.
          2/1       Filler for word boundary alignment
          3/1       Number of prompts
          4/1       Length of response to first prompt      (Note 1)
          5/?       Text of response to first prompt        (Note 1)
          ?/1       Length of response to second prompt     (Note 2)
          ?/?       Text of response to second prompt       (Note 2)

  Note 1 - Present only if number of prompts is one or two

  Note 2 - Present only if number of prompts is two
```

7.5.6.11 OISFMS.

This routine places the text of an operator request into the MAILBOX message buffer. It also builds the other invariant fields of such a message -- user ID, time, response mark and length.

7.5.6.12 OISMBX.

This is a service routine that sends the text of an operator request to one user or to the appropriate users in UIDT. This includes any user who asked to see all requests or whose user ID matches the user ID for the operator request being processed. OISMBX has two arguments, both of which are pointers. The first argument points to an entry in ORT. The second argument, which is optional, points to an entry in UIDT. If the second argument is supplied, the general text of the request is sent to the specified user. Otherwise, the general text is sent to all interested users in UIDT.

Another function performed by OISMBX is cleanup of UIDT. OPERATOR is not notified when a user logs off. Before the message is formatted for MAILBOX, the system PDT list is searched for the PDT associated with the station number in the UIDT entry. If the JSB pointer in the PDT is not the same as the JSB pointer in the UIDT entry, the user is no longer logged on at that terminal, and the UIDT entry is deleted.

7.6 OPERATOR INTERFACE TASK

The name of the operator interface task is XOI. The Pascal program in XOI is named XOITSK. XOI operates in the following two modes:

* When the system operator's terminal is dedicated to operator activities, the task provides an interface directly to OPERATOR for displaying messages and two functions, respond to operator request and kill operator request.

* When the user does any other operator activity, the task simply formats a message to OPERATOR, writes it to .S$OPER and processes the reply buffer. The user receives the text of operator requests through MAILBOX and SCI.

7.6.1 Invoking XOI.

XOI is bid by SCI with a PARMS list that contains the information required to write a message on .S$OPER for OPERATOR services. The PARMS list includes an opcode, which is not the same as the OPERATOR opcode. The specific elements in the PARMS list are discussed in the paragraphs on detailed design of the operator interface task.

7.6.2 Data Structures and Variables.

The operator interface task data structures and variables are as follows:

* DISPATCH. Internal code for the function to be performed. This is initialized based on the PARMS list, but in certain circumstances, it is modified during execution.

* READPEND. Boolean variable that indicates whether or not a read is active on .S$OPER

* XOIACTIVE. Boolean variable that indicates whether the terminal is dedicated to operator activities

* XOIMODE. Boolean variable that indicates whether or not this task represents the system operator

The following error variables are maintained by XOI:

* MSGNUMBER. Message number

* VARTEXT. Variable text for error message

* CONDCODE. Condition code. The only codes returned by XOITSK are the following:

  - >8000. Not an irrecoverable error

  - 0. Normal

CONDCODE is set, using the constants NORMAL and NONFATAL, which are defined in XOICONS.

The synonym $XOI$MEN is used to communicate with SCI regarding display of a menu. The command procedure ROM examines the synonym. If it is defined, ROM executes a .MENU primitive to to suppress the normal display of a menu. This preserves the screen displayed by XOI.

7.6.3 Initialization.

Initialization of XOITSK is done in the routine XOIINT. The routine sets up IRBs and pointers, prepares message and character buffers for later use, opens the channel S$OPER and stores a false value in the flags XOIMODE and READPEND.

7.6.4 Major Loop.

The major loop of XOITSK is repeated as long as XOIMODE is true.

At the top of the loop, XOISET is called to do the following:

* Checks for batch mode. This is accomplished by calling R$STAT to determine the state of the session. If it is batch, processing is aborted.

* Stores the station ID. This information is returned by R$STAT.

* Sets DISPATCH. This requires translating the parameter on the PARMS list to the proper OPERATOR opcode.

* Clears the error variables CONDCODE and MSGNUMBER.

The next portion of the loop is a case statement based on DISPATCH. The appropriate processor is invoked. If, after this processing, XOIMODE is true, XOI suspends by a call to R$WAIT. XOIMODE is set to true only when OPERATOR establishes the user as the system operator and is set to false when OPERATOR removes the user from the system operator designation. (In terms of SCI command procedures, XOI sets XOIMODE and QOI resets it.)

7.6.5 Termination.

When XOIMODE is false, XOITSK exits the major loop and closes the LUNOs for the terminal and for S$OPER. R$TERM is then called to terminate the task. The task also terminates (through R$TERM) if an SVC error occurs with regard to the read message on S$OPER or the keyboard read.

7.6.6 Error Processing.

XOITSK maintains error variables and reports errors through the Pascal interface routines R$TERM, R$WAIT and R$CMSG.

7.6.7 Detailed Design.

When XOI is bid, the PARMS list always contains the following three elements:

1. Pascal stack parameter - A value of 1000 is sufficient.

2. Pascal heap parameter - A value of 1000 is sufficient.

3. A parameter to indicate what service is desired

The third parameter passed from SCI to XOI is not the same as the opcode that is passed from XOI to OPERATOR. Both sets of values are shown in Table 7-1.

The fourth element varies with the service requested, as follows:

* XOI, QOI, KOM - No fourth element required

* COM - General text of the request

* ROM - Message selection (ALL or MY)

* LOM - Listing access name

* ROR, KOR - Request ID

COM has a fifth parameter, the operator interface channel pathname. This is .S$OPER, or <sitename>:.S$OPER, if the message is being sent to a network site.

Table 7-1 OPERATOR Opcodes

OPERATOR Opcode	Action	PARMS List Code	SCI Command
1	Creates operator request	8	COM
2	Starts relaying operator requests (through MAILBOX)	1	ROM
3	Stops relaying operator requests (through MAILBOX)	2	KOM
4	Lists operator requests	3	LOM
5	Designates user as system operator	4	XOI
6	Removes user as system operator	5	QOI
7/8	Responds to operator request	6	ROR
7/8	Kills operator request	7	KOR
7	Returns text and prompts of operator request		
8	Processes operator response to request		

The format of messages written on .S$OPER is discussed in the paragraph on detailed design of the processor that formats the message. The first byte is the OPERATOR opcode in all messages. The second byte contains flags. The format of the flags byte is detailed in Table 7-2 and is referred to as FLAGS in the detailed discussion of each message format.

Table 7-2 FLAGS Byte of .S$OPER Message

Bit(s)	Description
0	For use by relay operator messages (ROM) request. 1 - All operator messages 0 - Only messages associated with this user ID
1,2	Response type (for ROR) Bit 1 / Bit 2 / Type 0 / 0 / Positive 0 / 1 / Negative 1 / 0 / No response
3	Response required (for use with general operator requests) 0 - No 1 - Yes
4	For use with general operator requests 0 - User ID for this request is specified in in the JSB of the task that generated the message. 1 - User ID to be associated with this request is specified elsewhere in this buffer.
5-7	Reserved, set to zero

7.6.7.1 XOIXOI.

Procedure XOIXOI formats a message to OPERATOR requesting that the user be designated the system operator. If the request is granted, XOIXOI writes the following text to the system log:

```
userID - STxx BECAME SYSTEM OPERATOR
```

where:

userID and STxx (station number) identify the user who is designated system operator.

XOIXOI handles all I/O between the operator terminal and OPERATOR until the operator indicates, by pressing the CMD key, that the terminal is no longer to be dedicated to the system operator interface.

The Boolean variables XOIMODE, XOIACTIVE and READPEND are used throughout the procedure.

XOIXOI formats a message to OPERATOR and writes it to .S$OPER. If the reply buffer contains a nonzero value in the error code field, the request was not completed successfully. The message number, condition code, and variable text returned by OPERATOR in the reply buffer are stored in the error variables and XOIXOI is exited.

If the error code is zero, the user is now the system operator, and XOIMODE is set to true. The user's display is cleared. At this point, the terminal is dedicated to system operator functions. Two things can happen -- either an operator request can be written to .S$OPER by another user, or the system operator can initiate activity with OPERATOR by entering operator commands from the keyboard. XOIXOI must respond to whichever of these two events occurs first. The event synchronization facility of the operating system is used. An initiate read on .S$OPER is event zero, and an initiate read on the keyboard is event one. After the display is cleared, if READPEND is false, an Initiate Event SVC is issued for event zero and READPEND is set to true.

An Initiate Event SVC for reading the keyboard is issued, and XOIACTIVE is set to true.

XOIXOI enters a loop that is continued as long as XOIACTIVE is true. The loop consists of waiting for an event to occur, and processing the data associated with that event.

```
LOOP: DO while XOIACTIVE
        Issue Wait for Event SVC (either 0 or 1);
        IF the event is the channel read
           THEN IF SVC error
                   THEN Terminate through R$TERM;
                IF no channel error (in reply buffer)
                   THEN Abort keyboard read event;
                        Display data from channel read;
                Initiate Event SVC for channel read;
        IF keyboard read is complete (always true)
           THEN IF SVC error
                   THEN IF not due to intentional abort
                           THEN Terminate through R$TERM;
                   ELSE Call XOIKEY to process data from read;
                IF XOIACTIVE
                   THEN Initiate Event SVC for keyboard read;
END LOOP; (XOIACTIVE)
```

The test for keyboard event complete is superfluous because it either is the event that occurred, or it has been completed by the abort issued in processing the channel read complete.

Procedure XOIKEY processes keyboard input. All keys are ignored except F4, F5 and CMD. F4 is processed as a respond to operator request, and F5 as a kill operator request. If CMD is the event key, XOIKEY resets XOIACTIVE.

When the input is F4 or F5, XOIKEY sets variables as though XOI had been invoked by SCI to process a ROR or KOR request, respectively. The processing of the request is handled in XOIKEY, with calls to XOIROI, which processes both transactions.

Before returning to XOITSK, XOIXOI sets the variable text, message number, and condition code variables to indicate no errors.

The message format is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=5)
1/1	FLAGS
2/2	Station ID

7.6.7.2 XOIQOI.

XOIQOI formats the message to remove the user from designation as the system operator. If XOIMODE is false, the message number and condition code variables are set to indicate that the request is not allowed.

Otherwise, a message is formatted and written to .S$OPER. If a nonzero error code is returned in the reply buffer, the error variables are set to indicate an internal error message number, and a recoverable error condition code.

An error code of zero indicates success, and the following text is written to the system log:

```
userID - STxx QUIT AS SYSTEM OPERATOR
```

where:

> userID and STxx (station number) identify the former system operator.

If a read on .S$OPER is pending, the read is terminated and READPEND is set to false. (OPERATOR aborted the read while processing the message.) XOIMODE is set to false and error variables are set to indicate no errors.

The message format is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=6)
1/1	FLAGS
2/2	Station ID

7.6.7.3 XOICRM.

XOICRM formats a general operator request message for .S$OPER (SCI command COM). The kind of request generated is limited in that no prompts are allowed, and no time-out may be specified.

R$PARM is called to obtain the channel pathname and the message text from the PARMS list with which XOI was bid. The message is formatted and written to the specified pathname. When the reply is received, the XOITSK error variables are set to indicate whether an internal error or no error occurred.

The general operator request message format is as follows:

Offset/ Byte Length	Description	
0/1	OPERATOR opcode (=1)	
1/1	FLAGS	
2/2	Job ID to be used rather than ID of job from which the request originated.	
4/1	Time-out count (minutes)	
5/1	Number of prompts	
6/1	Length of general text	
7/?	General text	
?/1	Length of first prompt	(Note 1)
?/?	Text of first prompt	(Note 1)
?/1	Length of default for first prompt	(Note 1)
?/?	Text of default for first prompt	(Note 1)
?/1	Length of second prompt	(Note 2)
?/?	Text of second prompt	(Note 2)
?/1	Length of default for second prompt	(Note 2)
?/?	Text of default for second prompt	(Note 2)

Note 1 - Must be provided if number of prompts is one or two

Note 2 - Must be provided if number of prompts is two

Prompts, time-out, and response required are not available through XOICRM. These fields are utilized by tasks that write a message to .S$OPER with those options specified. (For example, the Spooler subsystem writes general requests with prompts, time-outs and response required.) The XOI user interface provided for creating an operator message is not designed to deal with the complexities of waiting for a response.

7.6.7.4 XOIROM.

XOIROM formats and writes a message on .S$OPER to request that OPERATOR place the user's ID in UIDT and relay messages to the user through MAILBOX, rather than directly to the terminal. If the error code in the reply buffer is nonzero, the error variables are set and XOIROM processing ends.

If no error is returned from OPERATOR, XOIROM tests the XOIMODE and READPEND variables to determine if a read on .S$OPER should be displayed. (OPERATOR aborts any read on the channel when the system operator quits or issues the ROM command. If, however, the read has completed, the data is in the reply buffer, and has not been displayed. The ORT entry is flagged as having been sent to the operator, and is not routed to the user through MAILBOX. Therefore, the request must be displayed now if the operator is to see it at all.) The request is displayed and READPEND is set to false. Notice that XOIMODE remains true, even though the system operator's terminal is no longer dedicated to system operator activities.

XOITSK error variables are set to indicate normal processing.

The format of the message is as follows:

Offset/ Byte Length	Description
0/1	OPERATOR opcode (=2)
1/1	FLAGS
2/2	Station ID

7.6.7.5 XOILOM.

Procedure XOILOM produces a file containing a list of pending operator requests. The pathname of the file in which to write the information is the fourth element on the PARMS list with which XOI is bid. The access name is obtained through R$PARM, and R$OPEN is called to open the file. If an error occurs, the error variables are set and processing ends.

After the successful open, the message is formatted and written on .S$OPER. If the error code in the reply buffer is nonzero, the error variables are set to reflect the kind of error, including the "error" of no pending operator requests. The user-specified file is closed by call to R$CLOS, with the parameter to specify that the file not be displayed. XOILOM processing ends.

The remainder of XOILOM consists of transferring the contents of the temporary file created by OPERATOR to the file with the access name provided by the user. The write operations are accomplished by calls to R$WRIT and R$WEOL. The file is closed by call to R$CLOS, with the parameter to specify that the file be displayed.

The format of the message is as follows:

Offset/ Byte Length	Description
0/1	OPERATOR opcode (=4)
1/1	FLAGS
2/2	Station ID

7.6.7.6 XOIKOM.

XOIKOM formats and writes to S$OPER a message to request that OPERATOR stop sending operator messages to the user through MAILBOX. Error variables are set to reflect the results returned in the reply buffer.

The message format is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=3)
1/1	FLAGS
2/2	Station ID

7.6.7.7 XOIROI.

Procedure XOIROI formats a message to OPERATOR for responding to or killing an operator request. It is invoked directly when SCI bids XOI to either kill or respond to a request. It is also called by XOIXOI when the system operator uses a function key shortcut. Because it may be called either way, an argument is passed to indicate whether the operator request ID is in the PARMS list (invoked by SCI) or in a buffer (called by XOIXOI). The variable DISPATCH is used to determine whether the call is for a response or a kill.

The first step in either case is to format and write a message to .S$OPER, asking OPERATOR for the general text and prompts, if any, of the specified operator request. If a nonzero error code is returned in the reply buffer, the appropriate error variables are set and processing ends. Among the possible "errors" returned is that the request requires no response. In this case, however, the error processing is appropriate, because there is no further action to be taken on the request. An operator request that requires no response is informational only, and killing or responding to it are equivalent.

A case statement, based on the value of DISPATCH, is used to set up prompts and initial values to be displayed to the operator.

* For the response operation, the data returned by OPERATOR in the reply buffer is used. If prompt text(s) and initial value(s) are supplied, they are saved for

display, and the type response expected from the terminal is non-null. If the prompt count is zero, the default prompt is used and a YESNO type response is required. The character string used for the default prompt is the constant RIODFLT.

* For the kill operator request, a single default prompt is set up, with a YESNO response type required. The character string used for the default prompt is the constant KIODFLT.

XOISIO is called to do the I/O to the display. When control is returned from XOISIO, a second message is formatted for OPERATOR. This message indicates one of three kinds of operator response to the request currently under consideration:

* No response - Operator has looked at the request, but chose not to respond (that is, he or she pressed the CMD key rather than responding to the prompt). This response is also sent when the operator denies a request that has no prompts, but requires a response or when the operator decides not to kill a request after having started to do so.

* Positive - Request has been considered by the operator and the data, if any, is included in the message.

* Negative - Request should be deleted from the ORT because the operator killed it.

In the case of the positive response, the data supplied by the operator is put into the message.

XOISVC is called to write the message to S$OPER. If an error is indicated in the reply buffer, the appropriate error variables are set.

The format of the first message is as follows:

Offset/Byte Length	Description
0/1	OPERATOR opcode (=7)
1/1	FLAGS
2/2	Request ID

The format of the second message is as follows:

Offset/Byte Length	Description	
0/1	OPERATOR opcode (=8)	
1/1	FLAGS	
2/2	Request ID	
4/1	Filler for word boundary alignment	
5/1	Number of prompts	
6/1	Length of first response	(Note 1)
7/?	Data in response to first prompt	(Note 1)
?/1	Length of second response	(Note 2)
?/?	Data in response to second prompt	(Note 2)

Note 1 - Present only if number of prompts is one or two

Note 2 - Present only if number of prompts is two

7.6.7.8 XOISVC.

XOISVC issues all supervisor calls for XOITSK. The SVC block must be prepared, and a pointer to it passed as an argument to XOISVC. If an error is returned, XOISVC calls R$TERM to report the error and terminate the task. No local error variables are set when an SVC error occurs.

7.6.7.9 XOISIO.

Procedure XOISIO formats the data to be displayed, and handles user inputs in a manner that emulates the SCI user interface. The I/O to the device is handled through R$FMT and R$GKEY, the Pascal interfaces to S$FMT and S$GKEY, respectively.

All event keys not processed by S$GKEY are ignored except for the following:

* Up Arrow key - Starts over on prompt number one.

* ERASE INPUT key - Ignores any previous input. Reformats the display with prompts and initial values and gets ready to accept input for prompt number one.

* ENTER key - Uses all current data, if it is acceptable.

* RETURN, TAB, SKIP keys - Check the value returned, and if it is acceptable, go to the next prompt, if any.

The values supplied by the user are checked in function XIOVAL, which is called with an argument to indicate the type response required and a pointer to the string to be checked. It returns a value of true except in the following cases:

* The length byte for a non-null string is zero.

* The first character of a YESNO type is not Y or N.

* R$INT returns a nonzero error code for the character string that is expected to be an integer expression.

7.6.7.10 XOIDSP.

This routine displays the message returned upon completion of a Channel Read SVC to .S$OPER. It is called with a Boolean argument, SUSPEND, that indicates whether to set the synonym $XOI$MEN. The appropriate SVC blocks are built, and XOISVC is called to write the data to the display. If SUSPEND is true, the synonym $XOI$MEN is given a value of Y. This synonym is used in the command procedures. When $XOI$MEN has a non-null value, the command procedures execute a .MENU primitive to suppress display of the menu in the major loop of SCI.

7.7 USER ACCESS TO THE OPERATOR INTERFACE SUBSYSTEM

Users can access the operator interface using several OI$ routines. Routines OI$BGN, OI$COM, OI$END and OI$WAT are documented in the DNOS System Programmer's Guide.

7.8 INTERNATIONALIZATION

The following messages in procedure/module OISRPL of OPERATOR are coded in English:

* **** OI - REPLY TO ##

* TIMEOUT

* REQUEST DENIED

* REQUEST GRANTED

The following text in procedure/module XOIINT of XOITSK are coded in English:

* RESPOND TO OPERATOR INTERFACE REQUEST

* KILL OPERATOR INTERFACE REQUEST
* REQUEST ID
* KILL REQUEST?
* GRANT REQUEST?
* -STxx BECAME SYSTEM OPERATOR
* -STxx QUIT AS SYSTEM OPERATOR

7.9 MAINTENANCE OF THE OPERATOR INTERFACE SUBSYSTEM

The user interface portions of the operator interface subsystem are designed to be compatible with SCI. In order to maintain this compatibility, the following things must be done:

* The S$ routines S$GKEY and S$FMT are linked into XOI. When changes are made in these routines, XOI must be relinked and reinstalled, in order to keep it compatible with SCI.
* If SCI changes the way the following keys are processed, an equivalent change is required in the XOITSK procedure XOISIO:
  - Up Arrow key
  - ERASE INPUT key
  - RETURN key
  - ENTER key
  - TAB key
  - SKIP key

SECTION 8

SPOOLER

8.1 OVERVIEW

The Spooler subsystem is the interface between users and output devices. The functions performed by the Spooler subsystem include the following:

* Maintains output queues

* Schedules and starts output devices

* Halts and resumes output at devices

* Services user requests to do the following:

  - Place an output request on a specified queue

  - Modify the priority, form or destination of a request already on an output queue

  - Display the status of devices and/or queues

  - Modify spooler device attributes

The Spooler subsystem is designed to prevent unauthorized deletion or modification of output requests, yet give a user complete control of the requests he or she initiated. It is written in Pascal and assembly language.

8.2 STRUCTURE

The Spooler subsystem consists of tasks that execute in the spooler job and tasks that execute in the user's job.

8.2.1 Tasks in the Spooler Job.

The Spooler job is created during IPL. The system initialization batch stream bids the Spooler device scheduler task (SP$DST) as the primary task. The XJ SCI command procedure specifies the task bid parameters for SP$DST. The leftmost 8 bits of Bid Parameter one (PARM1) are flags. The leftmost bit is normally set to 0, but may be set to 1 to specify that files are to be printed in priority order without regard for the currently mounted form. The other seven flags are reserved. The rightmost 8 bits of Bid Parameter one (PARM1) specifies the number of class name records to be created in the Spooler queue file, where each class name record will support 48 class name entries. Bid Parameter 2 (PARM2) specifies the number of device table records to be created in the queue file, where each device table record supports 12 device name entries. Because SP$DST changes the state of line printers, it must be a software-privileged task. SP$DST is privileged, software-privileged, and is not replicatable.

The following tasks are bid by SP$DST and terminate upon completion:

* SPINIT - Initialization task that is bid when SP$DST is first activated. It is neither reentrant nor sharable. SPINIT must be a software-privileged task and is not replicatable.

* xxWRITER - A task that performs an output request. xx indicates the device type. (For example, LPWRITER spools lines of data to a printer, with a device name that starts with the characters LP.) The writer task signals SP$DST when the output request is complete. Writer tasks are replicatable.

8.2.2 Tasks in the User's Job.

The following tasks are bid in the user's job and terminate upon completion:

* SPTASK - Allows a user task to route output to a logical name rather than to a device. SPTASK generates a request to place the spooled output (temporary file) on the appropriate queue for output service.

* PF - Interfaces with SCI and formats the information supplied by the user to generate the following SCI requests: Print File (PF), Halt Output (HO), Kill Output (KO), Modify Output (MO), Resume Output (RO), and Modify Spooler Device (MSD); finds a specified device entry in the Spooler queue file and set a synonym for the device's associated class names. This synonym will be the default value issued for the class names prompt of the MSD command. This prevents a user from inadvertently destroying other user's class names.

* SOS - Displays the requests queued for output and the status of Spooler devices. The output of this task is documented in the DNOS System Command Interpreter (SCI) Reference Manual with the SOS command.

8.3 COMMUNICATION AMONG SPOOLER TASKS

SP$DST communicates with other tasks through channels, task bids and semaphores.

8.3.1 Channels.

Three channels provide the interprocess communication required by the Spooler subsystem.

8.3.1.1 .S$DSTCHN.

.S$DSTCHN is the global master/slave channel owned by SP$DST. All requests for output service on spooler devices must be requested with a message to SP$DST on this channel.

All messages to SP$DST must be written with a request for reply. This is enforced in the code, and messages are ignored if no reply is specified. The length of the reply buffer must be greater than or equal to eight.

The PF task (in the user's job) sends formatted messages to SP$DST on .S$DSTCHN. Each message contains a spooler request code and the information required to perform the function indicated by the code. Details of the format are shown in the section of this document entitled Data Structure Pictures. The spooler message format is named SPM.

Most of the data is information needed to add, change or delete a request on the queue. The first two bytes, however, are an opcode (used by SP$DST to control flow) and a byte in which an error code can be returned by SP$DST to the originator of the message.

IPC passes open and close I/O operations to SP$DST as the channel owner. No task is allowed to open the channel with exclusive access. Servicing of close requests to the channel is not altered. SP$DST issues a Master Write SVC indicating that no error occurred.

The opcodes and the function each represents are as follows:

Code	Function
0	Writer task has completed or terminated
1	Adds an entry to an output queue
2	Halts output
3	Resumes output
4	Kills output
5	Modifies a previous request
6	Modifies attributes of a Spooler device
7	Checks validity of a Spooler device or class name
8	Find file name, given Spool ID
20	Maintains current copy count for a print request
21	Fake modify attributes request (used only by LPWRITER)
22	Fake modify output request (used only by LPWRITER)

8.3.1.2 .S$ACCCHN.

This job-local symmetric channel is owned by the accounting log task, LGACHN, and is used by SP$DST to place accounting information in the accounting queue when output service is completed.

The format of the message is shown in the Data Structure Pictures section of the DNOS System Design Document. The message is the same as the accounting record contents, ACC.

8.3.1.3 .S$SPOOL.

This task-local master/slave channel is owned by SPTASK. When a user assigns a LUNO to a logical name created with resource type SP, the channel .S$SPOOL is established as the route through which data is passed from the user's job to a temporary file.

The channel has associated parameters from the user's assign logical name process. Those parameters and their meanings are defined in Table 8-1. SPTASK requires that the logical name be created with all 7 parameters and that the parameters are specified in numerical order.

When the .S$SPOOL LUNO is released, SPTASK sends a print file message to SP$DST.

Table 8-1 Spooler Parameters

Parameter Name	Parameter Number
ANSII Format	00
Banner Sheet	01
Number of Lines/Page	02
Number of Copies	03
Forms	04
Device/Class	05
Spooler Logical Name	06

8.3.2 BID Statements.

Tasks are bid by SCI and by SP$DST using an appropriate SVC. The elements of the PARMS list passed with the bid are described in the detailed discussion of each task in the following paragraphs.

SCI command procedures bid PF and SOS, each with a PARMS list.

LPWRITER and SPINIT are bid by SP$DST.

The system task IOU bids SPTASK (the owner of the channel .S$SPOOL) as part of processing an assign LUNO to a logical name with resource type SP. The user's assign LUNO IRB is passed to SPTASK. It contains a pointer to the parameter list specified when the logical name is created by the user.

8.3.3 Semaphores.

SP$DST and LPWRITER communicate using semaphores. The semaphore concept is discussed in the <u>DNOS Supervisor Call (SVC) Reference Manual</u>. The semaphore is used to coordinate requests for halting, resuming, killing and modifying output at a device.

Each device that is available to the Spooler subsystem is assigned a unique job-local LUNO during the spooler initialization process, or when the device is made available to the Spooler subsystem via the MSD command. The LUNO number is known by both SP$DST and LPWRITER. This number is used as an index into the semaphore data structure (that is, the LUNO number is the same as the semaphore number). Since LUNOs are unique, a unique semaphore is referenced with regard to each device.

Semaphores are given initial values during SP$DST initialization.

8.4 DEVICES

Devices can be dynamically allocated and deallocated to the Spooler subsystem. The following functions are provided:

* Adds and deletes spooler device entries.

* Changes the device availability to Spooler.

* Modifies the set of class names associated with the specific device. Class name usage allows the user to define an output class, composed of a set of devices, and allows the Spooler subsystem to dynamically select an available device from that set.

* Specifies the form currently mounted on the device.

* Allows devices to be specified as available exclusively to the spooler or available to the spooler, but to be shared with other programs, or as queue only.

Output requests directed to a Spooler device that is currently defined as queue only are queued for future service when the device becomes available. This allows the installation to take devices from the Spooler on a temporary basis. The Spooler subsystem has exclusive access to any device so specified. A device may be usable by the Spooler but be specified as a shared device. The Spooler will contend with other users for this type of resource.

The Spooler subsystem interfaces with the operator interface subsystem when forms are to be changed on a device. The name of the last form mounted on each device is maintained in the Spooler device entry table.

8.5 THE QUEUE FILE

A major design objective of the Spooler subsystem is to maintain the integrity of the print queues in the event of a system crash or intentional stopping of the system. This is accomplished by keeping a disk file that contains information about Spooler devices, the status of each of those devices, class names associated with the devices, and information about each output request that has not yet been serviced or that is being serviced. The queue file resides in a directory on the system disk, .S$SDTQUE. The name of the queue file is the same name as generated operating system that is currently executing.

In order to minimize disk accesses, the logical record size is large (768 bytes). The queue file is an unblocked relative record file.

The Spooler maintains two types of queues for output requests -- class name queues and device queues. As mentioned earlier, each installation can edit the initialization batch stream .S$ISBTCH to specify the desired number of class name and device table records. The default values for the standard Spooler queue file are one class name record and one device table record. This gives the user the availability of 48 class names and 12 devices.

The task SP$DST has exclusive write access to the queue file. Other tasks access the file for reading only.

The structure of .S$SDTQUE is shown in Table 8-2.

Table 8-2 Structure of .S$SDTQUE

Record Number	Contents
1	Header record containing the name of the file, version number, the number of class name records in the file, and the number of device table records in the file.
2 to i	Class name records, each containing 48 class name entries
i+1 to j	Device table records, each containing 12 device entries entries
j+1 to 65535	Blocks of output requests. Each record has space for six queue entries.

8.5.1 Class Name Table (CNT).

The records 2 to i of the Spooler queue file contain class name table (CNT) entries. The organization and format of each CNT entry is shown in the section of this manual entitled Data Structure Pictures. The following information is maintained about each of the class name queues:

* Class information:

  - Number of devices that use this class name

- Status of the class (active, deleted, halted, etc.)
- Character string name of the class

* Queue header:
  - Record number, offset within that record
  - Priority of the request

8.5.2 Spooler Device Table (SDT).

The device table records of the Spooler queue file contain Spooler device table (SDT) entries. SDT entries contain information about each device known to the Spooler, and a queue anchor to the requests waiting specifically for the device. The data structure picture SDT in the last section of this manual shows the organization and format of each entry in this record.

The following information is maintained for each of the devices:

* Name of the device
* LUNO assigned to the device, if the device is currently usable by the Spooler subsystem
* Status of the device (active or deleted)
* Pointers for as many as six class names with which the device is associated; the pointers consist of a (class name record number, index into the class name record) pair.
* Name of the form currently mounted on the device
* Device type (byte) and flags from the PDT
* Flags to indicate the state of the spooler device:
  - Available exclusively to Spooler?
  - Shared device
  - Halted?
  - Busy? (Set if a request is active at the device.)

The following information is maintained about the queue header for each device:

* Record number and offset within the record

* Priority of the request

Information about the output request that is currently active (not on a queue) is maintained for each device.

The following information about the request, if any, that is active on a device is maintained for each device:

* Priority of the request

* Record number and offset within the record

* Number of units to page forward or backward (used only with Resume Output command)

* Flags used to communicate with the writer task

8.5.3 Queue Records.

The remaining records consist of blocked output requests. A request occupies 114 (decimal) bytes, and a record contains a maximum of six requests. The organization and format of each SDQ entry (request) is shown in the section of this manual entitled Data Structure Pictures.

Queue entries are chained forward, in descending job priority order. (The highest priority in the system is 0 and the lowest is 31. Thus, the queues are chained in ascending numerical priority order.) The initial output priority assigned to a request is the job priority of the job that generated the request.

The anchors for these queues are the CNT and SDT entries. Entries are not on multiple queues, and an entry that is active on a device is not on any queue. (Pointers to the active request reside in the device table entry for the device on which the request is active.) If the user requests that a multifile or concatenated file be output, the SDQ entry is the header for a linked list of entries that contain pathname information necessary to process the request.

There are two kinds of entries in the request queue. The SDQ data structure picture shows the format of both types. The most common entry is a queue entry. It contains information for starting an xxWRITER task. The second type is a continuation entry. It contains little more than names of additional files that are to be output as part of the same request. Every continuation entry that exists is associated with a queue entry.

The following information is maintained for both types of entries:

* Status of the request (active or deleted)
* Record number and offset to a continuation entry, if any
* Queue chaining information:
  - Record number and offset of the next request
  - Priority of the next request

8.5.3.1 Queue Entries.

A queue entry carries the following additional information:

* Information about the origin of the request:
  - User ID
  - Job ID
  - Job name
* Name of the device or class specified
* Details of the request:
  - Priority
  - Number of copies
  - Lines per page
  - Form to be mounted on device
  - Spooler ID of this request
  - Pathname of the first file to be output
  - Whether or not to print a banner sheet
  - Whether queued for a device or for a class name
  - Delete file after output flags
  - ANSI flag

8.5.3.2 Continuation Entries.

The continuation entry is used only when the request is to print a logically concatenated set of files. It contains status, queue chaining, and continuation information, the same as a queue entry. The remainder of the record contains a one-word count that is the number of additional pathnames, followed by the pathnames, each in the following format:

Ncc...cc

where:

N	is the number of characters in the pathname.
cc...cc	is the character string itself.

The number of pathnames in a continuation entry depends on the length of the pathnames. A total of 100 characters is available for packing pathnames in a continuation entry. The pathname list can be continued across as many SDQ entries as required.

8.5.4 Spooler ID Logical Names.

In the process of building a queue entry, SP$DST creates a logical name for the input pathname or pathnames supplied in the spooler message. This logical name is of the following format:

Snnnnn

where:

nnnnn is the ASCII representation of a five-digit (decimal) number.

The number is initialized during spooler initialization, and is incremented each time a request is added to a queue. The logical name created by SP$DST is the spooler ID displayed by the SOS task.

This logical name defines a single file or a concatenated file set. This definition simplifies the function of writer tasks. Under this arrangement, the writer tasks require no knowledge of the user's request; the writer task assigns to the spooler ID as defined in the queue entry. Because SP$DST has created this logical name, the operating system File Management subsystem builds the structures necessary to access the file(s). The writer task reads the file until an EOF or EOM is encountered.

8.6 DETAILED DESIGN OF SP$DST

Details of the device scheduler task, SP$DST, and its initialization task, SPINIT, are discussed in the following paragraphs.

8.6.1 Memory Data Structures.

Two major data areas are used by the Spooler subsystem:

* SPMSG - Segment containing the text of messages that are written by the Spooler to the system log. It is used by SP$DST. The structure is a table with 20 entries. The length of each entry is 50 characters.

* SPCOMN - Area containing run-time information about the queue file. SPINIT initializes most of this area for SP$DST, but some of the common area is used for queue positioning parameters internal to SP$DST. These structures are described in greater detail in Figure 8-1.

```
******************************************************************
*  DESCRIPTION OF THE SPOOLER IN-MEMORY DATA STRUCTURES.         *
*     THIS IS A PASCAL DEFINITION OF THE DNOS COMMON TEMPLATE    *
*     FOR SPDATA, AND IS MAINTAINED AS THE SPCOMN PROCEDURE      *
******************************************************************
SPDATA : PACKED RECORD
   CNTREC : CNR    ;           "CLASS NAME RECORD BUFFER AREA
       "
       " CNTREC IS THE INTERNAL BUFFER AREA FOR CLASS NAME RECORDS
       " FROM THE SPOOLER QUEUE FILE
       "
   SDTREC :PACKED ARRAY [ 1..12] OF SDT ; "DEVICE TABLE RECORD BUFFER
       "
       " SDTREC IS THE INTERNAL BUFFER AREA FOR
       " DEVICE TABLE RECORDS FROM THE SPOOLER QUEUE FILE
       "
   QREC1  : QR     ;           "QUEUE ENTRY RECORD BUFFER #1
   QREC2  : QR     ;           "QUEUE ENTRY RECORD BUFFER #2
       "
       " QREC1 AND QREC2 ARE INTERNAL BUFFER AREAS USED
       " TO BUFFER IMAGES OF RECORDS TWO THROUGH 255 OF
       " THE .S$SDTQUE FILE; QREC1 IS USED TO BUFFER THE
       " ENTRY BEING ADDED, DELETED, OR MODIFIED, WHILE
       " QREC2 IS USED TO BUFFER THE ENTRY THAT PRECEDES
       " THE ENTRY IN QREC1 IN THE QUEUE
       "
   HDRREC : HR     ;           "FILE HEADER RECORD
   MRAREA : MRA    ;           "MASTER READ BUFFER AREA
       "
       " MRAREA IS THE BUFFER AREA FOR THE SPOOLER MESSAGES
       " OBTAINED BY THE IPC MASTER READS AND WRITES ACROSS
       " THE .S$DSTCHN
       "
   SDFBLK : IRB    ;           "BLOCK FOR I/O TO SPOOLER QUEUE FILE
       "
       " SDFBLK IS THE I/O REQUEST BLOCK ( IRB ) USED TO
       " PERFORM I/O TO THE SPOOLER QUEUE RELATIVE RECORD FILE
       "
   MREAD  : IRB    ;           "MASTER READ/WRT SVC BLK
       "
       " MREAD IS AN I/O REQUEST BLOCK USED BY SP$DST
       " TO PERFORM IPC MASTER READS AND WRITES ACROSS
       " THE SPOOLER'S CHANNEL ( .S$DSTCHN )
       "
```

Figure 8-1 Spooler Data Structures (Sheet 1 of 4)

```
DEVIRB : IRB    ;             "DEVICE I/O BLOCK   ** 001 **
     "
     "
     "
     "
ACCIRB : IRB    ;             "ACCOUNTING CHANNEL ** 001 **
     "
     "
     "
     "
LNBLK  : S43    ;             "CREATE/DELETE LOGICAL NAME BLK
     "
     " LNBLK IS THE NAME MANAGER REQUEST BLOCK USED TO
     " CREATE SPOOLER LOGICAL NAMES IN THE PROCESSING OF
     " PF MESSAGES OR TO DELETE SPOOLER LOGICAL NAMES IN
     " THE PROCESSING OF COMPLETION MESSAGES FROM THE
     " WRITER TASKS
BIDSVC : S2B    ;             "BLOCK FOR BID TASK SVC
     "
     " THIS IS THE BID TASK SVC BLOCK USED TO BID THE DEVICE
     " WRITER TASKS AND SPINIT
     "
MAPPRG : S31    ;             "BLOCK FOR MAP NAME TI ID SVC
     "
     " THIS IS THE MAP NAME TO INSTALLED ID SVC BLOCK
     " THAT IS USED TO DETERMINE THE INSTALLED ID OF
     " SPINIT AND THE DEVICE WRITER TASKS PRIOR TO
     " THE ACTUAL TASK BID SVC
     "
SEMWAT : S3D    ;             "SEMAPHORE OPERATIONS SVC BLK
     "
     " THIS SVC BLOCK IS USED TO SEMAPHORE SIGNALLING
     " TO THE WRITER TASKS THAT IT SHOULD PREMATURELY TERMINATE
     " OR SUSPEND THE ACTIVE PRINT REQUEST.  THE MANNER OF
     " TERMINATION USED BY THE WRITER TASK IS A FUNCTION
     " OF STATUS FLAGS SET BY SP$DST IN THE APPROPRIATE
     " SDT ENTRY.
     "
```

Figure 8-1 Spooler Data Structures (Sheet 2 of 4)

```
MISFLG : PACKED RECORD       "MISCELLANEOUS FLAGS
   QTYP   : BOOLEAN;         "TRUE=DEVICE QUEUE ENTRY
   HOLFND : BOOLEAN;         "TRUE=AVAILABLE SPACE FOUND
   ACCOFF : BOOLEAN;         "TRUE=ACCOUNTING DISABLED
   DISABL : BOOLEAN;         "TRUE=DISABLE ALL CMDS THAT
   QFC    : BOOLEAN;         "TRUE=QUEUE FILE JUST CREATED
   FILL15 : 0..2047 ;
   END;
    "
    " MISFLG IS INTERNAL USE FLAGS
    "
SP$ID  : SID   ;             "SPOOL ID NAME AREA
    "
    " INTERNAL BUFFER USED TO BUILD AN ASCII SPOOL ID
    " LOGICAL NAME
    "
MSGADR : @SPM   ;            "ADDRESS OF MESSAGE TO DST
CURFRM : PACKED ARRAY [ 1..8 ] OF CHAR; "WORKING SPACE FOR FORM
SPLID  : WORD;               "SPOOLER ID
    "
    " INTEGER VALUE THAT REPRESENTS THE VALUE OF THE NEXT SPOOL
    " ID TO BE GENERATED
    "
CURCR  : WORD   ;            "CURRENT CLASS NAME RECORD
CURDR  : WORD   ;            "CURRENT DEVICE TABLE RECORD
FIRSTD : WORD   ;            "FIRST DEVICE RECORD NUMBER
FIRSTQ : WORD   ;            "FIRST QUEUE RECORD NUMBER
PRIOPT : WORD   ;            "PRINT BY PRIORITY OPTION FLAG
MAXQR  : WORD;               "MAX QUEUE RECORD
    "
    " REPRESENTS THE LAST RECORD NUMBER IN THE .S$SDTQUE FILE
    "
```

Figure 8-1 Spooler Data Structures (Sheet 3 of 4)

```
INIERR : BYTE;              "INITIALIZATION ERROR CODE
QUEOF1 : BYTE;              "OFFSET INTO QUEUE BUFFER ONE
QUERN1 : WORD;              "QUEUE RECORD NUMBER IN BUFFER 1
      "
      " QUERN1 AND QUEOF1 ARE A RECORD NUMBER/ENTRY OFFSET
      " PAIR INDICATING A PARTICULAR SDQ ENTRY IN THE
      " QREC1 BUFFER THAT IS BEING PROCESSED
      "
QUERN2 : WORD;              "QUEUE RECORD NUMBER IN BUFFER 2
QUEOF2 : BYTE;              "OFFSET INTO QUE BUFFER TWO
      "
      " QUERN2 AND QUEOF2 ARE A RECORD NUMBER/ENTRY OFFSET
      " PAIR THAT INDICATE THE  PARTICULAR SDQ ENTRY THAT
      " PRECEDES THE ENTRY INDICATED BY QUERN1/QUEOF1 PAIR
      " IN THE QUEUE CHAIN; IF QUERN2 = 0 AND QUEOF2 = >FF,
      " THEN THERE IS NO ENTRY ON THE QUEUE THAT PRECEDES
      " THE QUERN1/QUEOF1 ENTRY
      "
NDX     : BYTE;             "INDEX INTO RECORD
      "
      " USED BY THE SP$DST QUEUEING ROUTINES TO INDICATE
      " WHICH CNT OR SDT ENTRY IS BEING USED
      "
RESRV1 : BYTE;              "*** RESERVED ***
QFPN    : PACKED ARRAY [ 1..52 ] OF CHAR; "QUEUE FILE PATHNAME
```

Figure 8-1 Spooler Data Structures (Sheet 4 of 4)

8.6.2 Invoking SP$DST.

The spooler job is created when the system is initially loaded. The system initialization batch stream bids SP$DST as the initial task in the spooler job. If the spooler job is killed by the system operator, it can be restarted using the XJ command with SP$DST as the initial task. The user must remember to specify the identical task bid parameters that were last used in the initialization batch stream. If the Spooler initialization task finds differences in what is in the current queue file and what was specified in the task bid parms, the current queue file will be deleted and the task bid parms will be used to create a new queue file. This results in all device definitions and outstanding print requests being deleted.

8.6.3 Initialization.

A separate task, SPINIT, is bid by SP$DST to perform initialization functions for the Spooler subsystem. SPINIT calls procedure SPICHN, which deletes and recreates channels associated with the Spooler subsystem.

SPINIT calls SPIDTQ to assign, with autocreate, a LUNO to the spooler queue file. If no file is found, the Spooler queue file is created with no devices or class names defined. If the file is found, the active request at each device (if any) is placed on its original queue and the memory-resident variables are initialized to allow the restarting of output at spooler devices.

During reconstruction of the queues, SPINIT scans all the queues to determine the largest spool ID currently in use. (A given spool ID is associated with the same file across crashes.) SPINIT then terminates.

SP$DST assigns a LUNO to its channel, .S$DSTCHN, and the accounting channel, .S$ACCCHN, and schedules any device that is available and that has entries on its device queue or on one of its associated class name queues. SP$DST then issues a Master Read SVC to its channel, and is ready to process messages from users.

8.6.4 Major Loop.

SP$DST is the heart of the Spooler subsystem. It builds the prioritized queues from requests it receives, and schedules devices to perform the output requests on the queues. After initial setup, the flow is as follows:

```
DO UNTIL an irrecoverable error occurs;
   Clear previous error conditions;
   Call SPSCHD* to start the appropriate writer tasks;
   IF the write reply flag is set
      THEN Master Write to .S$DSTCHN;
   Master Read to .S$DSTCHN;
   IF the message is for SP$DST
      THEN CASE:Function code
          0:   SPDONE* - Writer task message
          1:   SPPFM*  - Output request
          2:   SPHOM*  - Halt output message
          3:   SPROM*  - Resume output message
          4:   SPKOM*  - Delete an output request
          5:   SPMOM*  - Modify an output request
          6:   SPMSDM* - Modify attribute(s) of a spooler device
          7:   SPVFY*  - Verify device or class name
          8:   SPFFN*  - Find file name, given spool ID
         20:   SPCPYC* - Maintain copy count for active entry
         21:   SPMSDM* - Fake MSD operation (used by LPWRITER)
         22:   SPMOM*  - Fake MO operation (used by LPWRITER)
      ELSE Report an error;
END;
```

* Name of the module that performs the function.

Various errors are reported to the user through SPERR, the error processing routine, but unless an error is catastrophic (see the subsequent paragraph on termination), control returns to the preceding loop.

8.6.5 Error Processing.

SP$DST and SPINIT process errors differently, as described in the following paragraphs.

8.6.5.1 SP$DST.

Error processing in SP$DST consists primarily of writing an error code in the user-specified reply buffer. Routine SPERR performs this function.

An error code of >E5 (bad call block) is placed in the second byte of the caller's request block if one of the following conditions is encountered:

* Caller does not specify write with reply
* The caller output character count (length of buffer) does not exactly equal the size of the SPM (template).
* Reply buffer supplied by caller is too short. Eight bytes is the minimum buffer length for returning the reply data.

If the caller attempts to open the channel .S$DSTCHN with any access privileges other than shared, >3B (unable to grant requested access privilege) is placed in the second byte of the caller's request block.

If an attempt is made to add a device or class name queue when the maximum number of such queues already exists, a message is written to the system log, and an error code is returned to the caller.

8.6.5.2 SPINIT.

Errors that occur in the initialization task are either ignored or cause abnormal termination of the spooler job. If SPINIT is unable to access a channel or a file that it needs, an error flag is set in SPDATA and the task terminates. The error flag causes SP$DST (and therefore the spooler job) to terminate.

8.6.6 Termination.

End-action is taken by SP$DST only when the task is terminated by the operating system (as a result of a Kill Job operation by the system operator).

SP$DST and SPINIT terminate when one of the following resources is unusable:

* IPC Channel .S$DSTCHN

* Spooler queue file

Routine SPQUIT handles termination. It takes the following steps:

1. Writes a message to the system log

2. Attempts to release all channels and associated LUNOs

3. Attempts to release the Spooler queue file

4. Issues an SVC to terminate SP$DST

8.6.7 Detailed Design.

The major loop of SP$DST consists of scheduling devices and processing formatted messages from various sources for various services. The modules that contain code to perform these functions are discussed in greater detail in the following paragraphs.

8.6.7.1 SPSCHD.

Procedure SPSCHD performs scheduling. The devices associated with a class name are scheduled for continuous operation as long as requests are queued. If a device or its associated class name queues contain requests, and the device is not halted, unavailable, or busy, SPSCHD selects a request to start on the device.

The priorities assigned the various requests are honored by the scheduler, regardless of the type queue in which the request is stored. Requests are normally selected from the device or class name queues according to requested print form first, and job priority second. This is to minimize operator forms mounting. Requests for the same form and having the same priority are processed on a first-in, first-out basis. Requests will be processed in priority order without regard to the requested print form if that option was specified when the spooler was started.

Once the next request to be serviced has been selected, SPSCHD removes the request from its queue, updating the appropriate queue header. The pointers for the active request are updated in the SDT. A writer task for the device is bid.

8.6.7.2 Queue File Space Management.

Ongoing management of space in the Spooler queue file is done by the SP$DST routines SPQADD and SPQDEL, which add and delete entries in the queues. The algorithm is designed to keep the queues compact, minimizing the number of disk accesses required.

The variable MAXQR is maintained in SPCOMN. Its value is the current number of records in the file.

Acquiring Space.

The queue file is created by SPINIT with the specified number of class name records, the specified number of device table records, and one (empty) queue entry record. It is expanded by SPQADD, one record at a time, as required to accommodate requests.

When a queue entry is ready to be placed in the file, the following algorithm is exercised to find space for it:

```
IF  There is a deleted entry in the current record
  THEN Write the new entry there;
  ELSE
      Starting with the first queue record, search
       (sequentially) for a deleted entry.
      IF a deleted entry was found
        THEN Write the new entry there;
        ELSE
          IF MAXQR = >FFFF
            THEN Report "space not available"
             and ignore the request;
            ELSE
                Increment MAXQR;
                Expand the file;
                Write the new entry in the new record;
ENDIF;
```

Releasing Space.

SPQDEL deletes entries from the queue records. The chain pointers are updated in the proper entries, and the queue entry as well as all associated continuation entries are marked deleted.

8.6.7.3 Writer Task Messages.

Device writer tasks send a formatted message on .S$DSTCHN to indicate to SP$DST that a request has been completed or terminated. Procedure SPDONE processes this message.

Seven conditions cause a writer task to generate a spooler message:

* Normal completion of the task. A termination due to an I/O error in the file being output is treated as a normal completion.

* The active request has been terminated by a Kill Output command.

* The active request has been terminated by a Modify Output command.

* Device error

* Request SP$DST to update (decrement) copy count for specified queue entry

* The operator responds positively to a forms mount request, thus requiring the currently mounted form name (in the SDT) to be changed.

* The operator responds negatively to a forms mount request, thus requiring the selected file to be requeued.

Flags in the device table entry and in the spooler message are used to determine what condition caused the writer task to send a termination message as follows:

* SDTFLG.SDFKIL=true means the task was killed by a kill output request. SP$DST signaled the writer task via a semaphore operation to terminate.

* SDTFLG.SDFTRM=true means the task was terminated during processing of a modify output message request. SP$DST signaled the writer task via a semaphore operation to terminate.

* SPMFLG.SDFDVE=true means that a device error occurred.

* SPMFLG.SPFABE=true means the device writer task took end-action.

SPDONE writes a record to the accounting channel, unless the flag SPDATA.MISFLG.ACCOFF is set to inhibit accounting.

SPDONE then determines the device to which the spooler message applies. This is done by matching the device name in the message with an entry in the device table.

If the writer is terminated by a Modify Output command, the active request is placed back on the proper queue by SPMOM. The only processing done by SPDONE is to reset the flag SDFTRM. This completes processing by SPDONE for the termination message generated due to a Modify Output command.

For those requests that specified multiple copies, the LPWRITER will, upon completion of each copy, send a message to the Spooler Device Scheduler Task. The message informs SP$DST to decrement the number of copies for the specified entry and update that entry on disk. In the event of a crash or intentional kill of the Spooler job, the original copy count will not be printed again; only the number remaining will be printed.

If the message is generated because of a device error, a Halt Output command is simulated. This consists of the following:

* Setting the halt flag in the device table entry

* Updating the device table record in the Spooler queue file

* Signaling the writer task, via semaphore, to halt and wait for a signal, and, again via semaphore, to resume output

In all other cases, the following processing is done:

```
IF the writer task ended abnormally
   THEN Write a message to the system log;
IF the flag SQFDAP (delete after print) is set and
 the file was successfully printed
 or the flag SQFDAL (delete always) is set
   THEN Delete the file;
Delete the queue entry;
Update the device table entry to show that no request is
 active at this device.
Reset the kill, busy and termination flags in the device entry;
ENDIF;
```

8.6.7.4 Output Request Messages.

Procedure SPPFM processes output request messages written from the user's job, for instance by PF or SPTASK. The process consists of the following steps:

1. Determining whether the request belongs on a device queue or a class name queue. If the device or class name is not found, an error is reported through SPERR and no further action is taken on the message.

2. Finding space in the appropriate queue and reporting the error if no space is available

3. Formatting the SDQ entry or entries (procedures SPBLDQ and SPCONQ)

4. Adding the entry to the queue (procedure SPQADD)

If a device known to the Spooler subsystem is not currently available, queueing of entries for that device continues. The requests are output whenever the device is made available to the Spooler subsystem. If a device is known to the Spooler, but is a shared device, the Spooler must contend with other tasks in the system for the use of the resource (device).

8.6.7.5 Kill Output (KO).

Either the system operator or the originator of an output request can issue a command to kill the output request. The PF task in the user's job writes the message to .S$DSTCHN that requests a kill output for the user. This message is processed by procedure SPKOM.

The initial test is to determine whether the user has requested that the entire queue or a single entry be removed. If the spooler ID supplied is ALL, the entire queue is examined for candidates to delete. After determining whether the name supplied is a device or a class name, the routine KILL_QUE is called to delete all output requests that the caller is authorized to delete.

In the event that the user identified a specific request (spooler ID), the active requests are searched first. If the specified request is found, SPKOM terminates the writer task, via semaphore operation, and updates the queue to delete the request. If the request is not active, it is waiting in the queue.

After the request is located, SPKOM calls SPOPCK to check the authority of the user to kill the request. SPOPCK returns a value of true if the user is either the system operator or the user who originated the request. If the user is authorized to kill the request, it is unchained and deleted. If the operator check is false, a privilege error indicator is returned.

KILL_QUE is a routine that processes an entire queue. Starting at the queue header in the class name or device table, each queue entry is read, and SPOPCK is called to check operator privilege. If the user has authority to kill the request, the entry is unchained and deleted. If the entry cannot be killed by this user, it is left on the queue. No privilege error messages are generated as it is legitimate for a user to kill all requests on a queue that belong to the user. The system operator is allowed to kill any request.

8.6.7.6 Modify Output (MO).

Procedure SPMOM processes this message type. This command allows the user to make various changes in entries that are already queued for output. The following items may be changed:

* Device name or class name - Removes the entry from its present queue and places it on the queue for the specified device or class name

* Form - Changes the form originally requested to the form now being specified

* Priority - changes the priority of an entry. The original priority is the priority of the requester's job, and can be changed to any value that is a valid job priority.

Valid job priorities are 0 (highest priority) through 31 (lowest priority).

SPMOM determines what is being changed by examining the value of the following three variables passed in the spooler message:

* SPMJPR - A value of >FF means there is no change in request priority.

* SPMDVN - A value of eight blanks means there is no change in device or class name.

* SPMFRM - A value of eight blanks means there is no change in form name.

The first test performed by SPMOM is to determine if this request involves changing the device/class. If so, the SDT or CNT entry for the specified device or class is located.

The second step is to locate the SDQ entry specified by the spool ID. The active devices are search first. If the entry is not found as an active entry, then the waiting queues are searched. If the specified spool ID is not found, an error is given. Otherwise, the location of the SDQ entry is noted along with whether the entry was active or not.

SPMOM then calls SPOPCK to verify that the requestor has the authority to make changes to the specified SDQ entry.

If the entry is not an active entry, it is unchained from the waiting queue.

If the entry is active and the only change specified is to numerically lower the priority (logically increase the priority), then the priority in the SDQ entry is updated and SPMOM returns normally to its caller.

If the entry is active and some other change is specified, then SPMOM signals the writer task to terminate.

Finally, SPMOM queues the entry to the specified device/class or, if none was specified, then to the device/class for which the entry was previously queued.

8.6.7.7 Modify Spooler Device (MSD).

This message type is processed by the procedure SPMDSM. Modify spooler device messages alter the entries in the device table and class name table. If an entry in the tables is not found for the specified device or class name, it is assumed that the new name is to be added.

In altering existing device or class name attributes, the following logic is exercised:

```
IF the delete device (SPFUSE) is set in the spooler message
   THEN
         IF the device is available to Spooler subsystem
            THEN Release the device;
         Mark the device deleted in the SDT;
   ELSE Update the entry with newly specified attributes;
ENDIF;
```

When new class names are specified for a device, the entire list of previously specified class names for that device is replaced with the new list.

When the device or class name specified does not match any entry known to the Spooler, the new information is used to construct an entry in the device table. A Map Task Name to Installed ID SVC is issued to ensure that there is a writer task for the specified device. If none is found, the device is not added and a name error message is returned through SPERR.

A device for which a writer task is found is added to the device table. Initially it has no request active and no requests queued. The form and device name are copied from the spooler message.

If the spooler mode indicates that the device is available, the routine SPDIAG is called to determine the status of the device. SPDIAG searches the system PDT list for the PDT of the device to be added. If no PDT is found, SPDIAG returns an error to the caller. The caller deletes the device table entry, when appropriate.

If the device is to be available exclusively to the Spooler, the device state is tested. Unless the device is online, or in the spooler state, the not available flag in the device table entry is set and processing is complete.

For devices found in the online state and are to be available exclusively to the Spooler, SPDIAG issues a special assign LUNO (an SVC that is reserved for system use) that gives the Spooler exclusive control of the device. If the device is shared, a regular Assign LUNO (op code >91) is performed. If an error is returned from the assign LUNO SVC, the not available flag in the device table entry is set.

Class names are added to the device table entry and both the class name and device table records are rewritten to the queue file.

8.6.7.8 Halt Output (HO).

SP$DST sets the halted flag in the device table entry and writes the updated entry to disk. If the HO request indicates an immediate halt and if a request is active on the device, SP$DST signals the writer task, via semaphore operation, to halt and wait for another signal, via semaphore operation, to resume output.

If the request does not specify an immediate halt, the halt output flag in the device entry table is set, but no semaphore operations are performed. After the active request is completed, the device is not scheduled again until the halt output flag is reset by a Resume Output command.

If the request specifies a class name rather than a device name, then the class is marked halted. This means that SP$DST will not schedule any more files in that class queue for printing on any devices until a Resume Output command is entered to reactivate the class queue. Any files currently printing at the time of the Halt Output command continue to print.

8.6.7.9 Resume Output (RO).

SP$DST resets the halted flag in the specified device table entry. The user-specified number of pages is stored in the queue entry for the request. A positive number causes pages to be skipped; a negative number causes pages to be reprinted.

SP$DST signals the writer task to resume execution via semaphore operation.

If the request specifies a class name rather than a device name, then SP$DST marks the class as not halted. This allows SP$DST to again schedule files in the specified class queue for printing on devices in the class.

8.6.7.10 Verify Device or Class Name.

The requested operation determines whether the specified device or class name currently exists in the SDT or CNT. SP$DST searches the CNT and SDT to find the device or class name specified. If it is not found, a name error message is reported through SPERR. Otherwise, no error is reported and processing ends.

The verify request is generated by the task that assigns spooler parameters, ASP, during the creation of a logical name with resource type SP. It is also used by SPTASK to verify that the device/class name specified is still valid.

8.6.7.11 Perform Copy Count Maintenance.

If a user requests multiple copies on a print request, the writer task sends a message to SP$DST when each copy completes. This message indicates that SP$DST is to reduce the number of copies for the active entry on the specified device.

8.6.7.12 Find File Name.

This message type is processed by the procedure SPFFN. SPFFN searches first the active devices for the specified spool ID. If the specified entry is not found as an active entry, SPFFN then searches the waiting queues. If the entry is still not found, an error is returned. Otherwise, the pathname of the file to be printed is returned to the requesting task via the reply buffer. The function provided by SPFFN is not used by any of the standard DNOS tasks. However, this functionality is provided for user programs to use if they need to.

8.6.8 Shared Modules.

The modules described in this paragraph are used by the tasks SPINIT and SP$DST. They must be linked into both tasks. The modules are as follows:

* SPDCNT - Marks the CNT entry specified by the calling parameter as a deleted entry. The queue headers are also deleted.

* SPDIAG - Assembly language routine that interfaces between the Spooler and the operating system PDT list. When the device is found in that list, a spooler job-local LUNO is assigned to it. The SDT entry is set to indicate that the device is not busy and is not halted.

* SPDSDT - Deletes the specified entry in the SDT. The device is marked deleted and the queue headers as well as the active request are altered to indicate that there is no request active, and none queued.

* SPIO - Routine that performs I/O requests for a specified record and record type.

* SPSDQD - Marks the specified queue entry deleted.

8.6.9 Internationalization.

The text of each message written to the system log is hard coded in English in module SPCOMN.

8.7 SPOOLER DEVICE WRITER TASKS

A spooler device writer task is bid when the device is found to be available, idle, and not halted, and an entry exists on the device queue or on one of its associated class name queues. In order to bid the writer task, SP$DST locates a task in the system utility program file, .S$UTIL, with the task name consisting of the first two characters of the device name followed by WRITER.

Device writer tasks have the following responsibilities:

* Requesting that SP$DST maintain the number of copies

* Maintaining the lines-per-page count

* Adding carriage control, unless the file has embedded carriage control

* Requesting the mounting of new forms

The writer task terminates after servicing a single request.

The only device currently supported by the DNOS Spooler is the line printer. LPWRITER services all line printers supported by DNOS.

8.7.1 Invoking LPWRITER.

SP$DST bids LPWRITER in the spooler job and passes information in the PARMS list. LPWRITER is passed 3 values: Bid parm 1 is XXYY, where XX is the shared luno assigned to the Spooler queue file and YY is an index into the device table record; Bid parm 2 is DDDD, where DDDD is the device table record number. This is where LPWRITER finds pointers to the output request that is currently active on the device.

8.7.2 Initialization.

Prior to printing a file, the Spooler queue file on disk must be opened and the device table read to obtain current information about the device. The queue entry for the request that is active on the device is then read into memory.

If the form currently mounted on the printer is not the form specified in the request, LPWRITER calls SPFORM to interface with the operator interface subsystem. No further action is taken on the request until SPFORM returns control to LPWRITER.

Following the call to SPFORM, the status of the device is checked to determine whether or not the active request has been killed. If so, LPWRITER closes the queue file, and formats and sends a message to SP$DST indicating that LPWRITER is done. LPWRITER then issues an SVC to terminate the task.

Procedure SPFORM interacts with the operator interface subsystem when forms need to be changed on a printer. A general operator request is written on .S$OPER, the global channel owned by the OPERATOR task. The format of the message is discussed in the section of this manual entitled Operator Interface.

SPFORM constructs a message with the following characteristics:

* Response required

* Job ID as the identifier

* Prompt count of one

* Message: MOUNT FORM = form name ON device name.

* Prompt: FORM MOUNTED

* Name of the requested form as the default value for the prompt

If the reply for the operator interface subsystem indicates that the specified job ID is not valid, the operation is tried again without specifying a job ID.

If the operator responds negatively, SPFORM sends a modify output request to SP$DST to tell SP$DST to requeue the file to the device or class for which it was previously queued.

If the operator responds positively, SPFORM sends a modify spooler device request to SP$DST to notify SP$DST that the operator has changed the form. If the operator specifies that the form mounted is one other than the form specified for the file, SPFORM then sends a modify output request to SP$DST to tell SP$DST to requeue the file.

Finally, SPFORM closes and releases its LUNO to the operator interface and returns to SPLPWT.

8.7.3 Processing a Print Request.

LPWRITER issues an Open File SVC for the device LUNO, prints a banner sheet if requested, opens the input file, and reads the first record. Carriage control information is extracted from the first record. The procedure SPCOPY prints one copy of the file or files. LPWRITER calls SPCOPY repeatedly for multiple copies.

SPCOPY is a loop that consists of the following:

* Writing a record to the device

* Testing the semaphore associated with this device. If SP$DST has signaled LPWRITER, the appropriate action is taken.

* Reading the next record from the input file

* Reporting input file I/O errors

This loop is repeated until the EOF is encountered in the input file or until the request is aborted.

When LPWRITER is signaled, the device table entry is reread and the following processing is done:

* If the termination flag SDTFLG.SDFTRM or the kill flag SDTFLG.SDFKIL is set, LPWRITER closes the input file, the device and the disk queue file, releases the associated LUNOs, sends a completion message to SP$DST, and issues an SVC to terminate the task.

* If the halt flag SDTFLG.SDFHLT is set, LPWRITER waits for a signal via semaphore operation indicating a resume output. The queue entry is read again and the page number is changed to reflect the page forward or page back information in the resume output message. The input file is repositioned, if necessary, and execution continues in the inner loop.

SPCOPY builds the SVC block to write a line to a printer. SPPRIO interfaces with the DSR. If the error code returned from the SVC is a device error, the write is retried as many as 10 times before a device error is returned to SPCOPY.

NOTE

The DSR error code 6 is assumed to be a device error. This is hard coded in SPPRIO. The DSR open error code >3B is assumed to be an open access privilege error. This is hard coded in SPPRIO.

8.7.4 Error Processing.

Routine SPLOGM writes various error messages to the system log. The messages are of the following format:

```
**** LPWRITER userID jobname ERROR ec:file/device
```

where:

userID	is the user's ID.
jobname	is the name of the job associated with the active request.
ec	is an error code.
file/device	is the name of the file or device on which the error occurred.

8.7.5 Termination.

Normal termination of LPWRITER occurs in the modules SPLPWT and SPCOPY and consists of the following steps:

* Closing the input file and releasing LUNO
* Closing the device LUNO
* Closing the Spooler queue file LUNO
* Calling SPLOGM to write an error message to the system log, if appropriate
* Formatting and sending a writer task completion message to SP$DST and waiting for the reply. This message contains the information SP$DST needs to build an accounting record.
* Terminating LPWRITER

A completion message is formatted and sent to SP$DST and the task is terminated. When the termination is because of an I/O error in the file being printed, an error message is written to the system log.

8.7.6 Internationalization.

In procedure SPFORM, the following English messages are constructed for the general operator request message:

```
MOUNT FORM=

ON

FORM MOUNTED
```

In SPLOGM, the fixed portion of the following message written to the system log is hard coded:

```
**** LPWRITER userID jobname ERROR ec:file/device
```

The block character definition for the banner sheet in SPCHAR is in English characters.

8.7.7 The Banner Sheet.

Pascal function SPBANN is the module that displays a user-requested banner sheet. The banner sheet is driven from the contents of the disk file .S$SDTQUE.S$BANNER. The banner sheet will examine the records of the disk file for command records. These command records and their indicated functions are:

* ≠JOB - Display user's job name in large block letters

* ≠USER - Display user's ID in large block letters

* ≠FILE - Display requested print file name

NOTE

If output is directed to a Spooler logical name, that name is printed in large block letters; otherwise a single line is displayed.

* ≠TEXT,CCCCCCCC - Display characters 'CCCCCCCC' in large block letters

NOTE

≠TEXT must be immediately followed by a comma or the record is ignored. The next 8 characters will be displayed in large block letters.

* ≠DATE - Display date and time

The standard banner sheet will consist of ≠JOB, ≠USER, ≠FILE, and ≠DATE command records. Any record that is not a command record will be echoed to the output device. Using this feature and the ≠TEXT command record, the user can easily use the text editor on .S$SDTQUE.S$BANNER to create a custom banner sheet.

8.8 SPTASK

SPTASK acts as an interface between a task in the user's job and a logical name created with spooler (SP) resource type. This allows user tasks to assign a LUNO to a Spooler logical name instead of to the actual device. When the LUNO is released, SPTASK writes a message to SP$DST on the channel .S$DSTCHN. The message requests that the file be output to the device or class name specified when the logical name was created.

SPTASK is written in assembly language. Functionally it performs exactly like the line printer DSR does.

8.8.1 Invoking SPTASK.

When a LUNO is assigned to a logical name with resource type SP, SPTASK is bid in the user's job by the operating system task IOU.

A unique SPTASK is invoked each time a LUNO is assigned to a logical name with resource type SP. For example, if a user has created a logical name SYSOUT as a Spooler resource type, and assigns two LUNOs to SYSOUT, there are two separate SPTASK tasks, two temporary files, and two separate output requests.

When SPTASK is bid, it is passed the IRB that is requesting an Assign LUNO operation to the user's logical name. See the Data Structure Pictures section of the DNOS System Design Document for details of the IRB format. Associated with the IRB is the set of parameters describing the print file options defined with the logical name. These will be used later.

8.8.2 Initialization.

During the initialization phase, SPTASK issues an SVC to get the job and task IDs in which it is executing. This information is used to construct a pathname in the following format:

```
.S$jjjjtt
```

where:

jjjj	is the ASCII representation of the internal hexadecimal job ID.
tt	is the ASCII representation of the internal hexadecimal task ID.

This name is used in the termination phase of SPTASK to rename the temporary file to which output has been spooled.

SPTASK then issues a Master Read to .S$SPOOL, the task-local channel across which all spooled output is received. If the first message on the channel is not an Assign, a message is written to the system log and SPTASK terminates.

When the Assign is received, SPTASK sends a Verify Device/Class Name message to SP$DST via the channel .S$DSTCHN to verify that the device or class specified as one of the parameters is still a device or class that is known to the spooler. If it is not, SPTASK puts an error >21 into the Assign LUNO call block, does a Master Write to complete the Assign LUNO processing, and then terminates.

If the device/class is valid, SPTASK issues an Assign LUNO for a temporary file with autocreate. If the file is opened without error, SPTASK then issues a Master Write to .S$SPOOL to complete the Assign LUNO processing.

8.8.3 Major Loop.

The major loop of SPTASK consists of Master Read, I/O to the temporary file, and Master Write. This loop is repeated until SPTASK receives a request to release the .S$SPOOL LUNO (that is, a release LUNO from the user task to its logical name).

8.8.4 Termination.

When the release LUNO is received, SPTASK writes a close EOF to the temporary output file. The file is then renamed .S$jjjjtt. A spooler message is constructed to place the file on an output queue. The message specifies that the file be deleted after it is printed. These parameters from the Assign LUNO IRB are used to build the Spooler message to request output service on the file.

SPTASK writes the message to .S$DSTCHN and waits for the reply. If an error is returned, a message is written to the system log.

NOTE

The task-local LUNO for .S$DSTCHN is specifically assigned to >77.

Having put the output request on a queue, SPTASK then begins termination processing. A Master Write to .S$SPOOL clears the channel. .S$SPOOL is then closed and the LUNO is released. .S$DSTCHN is closed and its LUNO is released. SPTASK then issues an SVC to terminate the task.

8.9 SHOW OUTPUT STATUS

SOS executes in the user's job. It displays items queued for output for each device and each class name. It also displays that status of each device. The user specifies the following options:

* All devices known to the Spooler subsystem, or a specific device or class, the status of that device, and the currently mounted form

* All requests or the requests queued for the requester's user ID, including the remaining number of copies to be printed

SOS is written in Pascal, and requires read access to the Spooler queue file.

8.9.1 Invoking SOS.

SOS is invoked by SCI with the following PARMS list:

1. Pascal stack parameter - A value of 1000 is sufficient

2. Pascal heap parameter - A value of 1000 is sufficient

3. Device/class name - Character string. A null character string for this variable is interpreted as a request that all queues be displayed.

4. User ID - Character string. A null character string for this variable is interpreted as a request that all entries in each queue be displayed.

8.9.2 Processing.

SOS opens the listing and Spooler queue files. The class name records and the device table records are sequentially read.

The CNT is searched for the specified class name. If it is not found, the name supplied is assumed to be a device name. First the device queues and then the class name queues are displayed.

8.9.3 Error Processing.

Errors in SOS are processed through UTCHEK, the SCI error reporting interface for Pascal.

8.9.4 Internationalization.

In the procedure SOPHDR, the following headings for the queue list are hard coded in English:

```
  DEVICE=XXXXXXXX
  STATUS= EXCLUSIVE
     OR = QUEUE ONLY
     OR = HALTED
     OR = SHARED
  FORM=FFFFFFFF
  CLASS NAMES=( )
  ST/     USER    FORM    SPOOL    LOGICAL NAME OR
  PRI      ID              ID         FILE NAME
```

8.10 PF

PF executes in the user's job. It accepts user information through SCI to format and send spooler messages to SP$DST, in the spooler job, for the following requests:

* Print Files
* Halt Output
* Resume Output
* Kill Output
* Modify Output
* Modify Spooler Device

PF is written in Pascal.

8.10.1 Invoking PF.

PF is bid by SCI and accepts parameters through the PARMS list. These positional parameters are as follows:

Position	Parameter
1	Pascal stack size
2	Pascal heap size
3	Command type (integer) 1: Print File command 2: Halt Output command 3: Resume Output command 4: Kill Output command 5: Modify Output at Device command 6: Modify Spooler Device command 7: Build synonym for MSD prompt
4	Spool ID
5	Device or class name
6	Class name list
7	Delete SDQ entry? (Y/N)
8	Form name
9	File pathname
10	ANSI? (Y/N)
11	Delete after print? (Y/N)
12	Banner sheet? (Y/N)
13	Lines per page
14	Number of copies (integer)
15	Page count (integer, used only by Resume Output command)
16	Immediately or at EOF (integer, used only by Halt Output command)
17	Priority (integer)
18	Available exclusive to Spooler? (Y/N)
19	Shared? (Y/N)

8.10.2 Processing.

The first part of PF performs processing common to all PF commands. The device or class name in the PARMS list is stored in the message buffer. If the form name PARM is blank, STANDARD is used in the message buffer. The spooler ID, if present, is saved. An SVC is issued to get the job name and ID, the user ID, and the job priority.

After processing the particular function code, PF assigns a LUNO to and opens .S$DSTCHN. The message is written to the channel. An error code is set, based on the reply received from .S$DSTCHN. The channel is then closed and the LUNO is released. UTPUER is called to return the error code and to terminate PF.

8.10.3 Error Processing.

Errors in PF are processed through UTCHEK, the SCI error reporting interface for Pascal.

8.10.4 Internationalization.

In PF, if no form is specified on the PARMS list, the English character string STANDARD is used.

SECTION 9

FILE MAINTENANCE UTILITIES

9.1 OVERVIEW

DNOS file maintenance utilities provide the following functions:

* Copy a hierarchical structure to another hierarchical structure
* Copy the data in a hierarchical structure to a sequential structure, and rebuilds a hierarchical structure from the sequential structure
* Verify the two kinds of copy operations
* List the elements of a hierarchical structure
* Map the contents of a disk volume
* Delete all elements of a hierarchical structure

File maintenance utilities are written in assembly language.

NOTE

Refer to the DNOS System Design Document and to the DNOS Systems Programmer's Guide for discussions of disk structures and files. The characteristics of those entities are integral to the design of the file maintenance utilities.

9.2 MOVE TASKS

A set of five tasks provides the following functions:

* Copy directory (CD) - Copies a hierarchical structure to a hierarchical structure

* Verify copy (VC) - Compares two hierarchical structures and reports the results

* Backup directory (BD) - Copies a hierarchical structure to a sequential structure

* Restore directory (RD) - Recreates a hierarchical structure from data previously copied to a sequential structure (by BD)

* Verify backup (VB) - Compares a hierarchical structure to backup data in a sequential structure and reports the results

Each of these tasks is a collection of common modules and unique modules in the DSC.DP.CD directory. In order to avoid confusion, the group of tasks is called the move tasks rather than the CD tasks, since CD is the name of one of the tasks.

Directory file structures and the files themselves are processed according to entries in the control file. If no control file is provided, all elements of the directory are processed, with the exception of temporary files, and certain system files, as noted in the paragraph entitled Design Concepts.

NOTE

The syntax and use of the control file is documented in the _DNOS System Command Interpreter (SCI) Reference Manual_.

Move tasks execute either interactively or in batch mode except when a multivolume medium is used. The message to mount the next volume must be written to an interactive terminal.

9.2.1 Design Concepts.

Each of these tasks is software privileged and replicatable. Exclusive access to all files is required. Once the file has been processed, however, the file is released.

Each move task is designed to process files that are elements of a DNOS directory structure. See the data structure pictures section of the _DNOS System Design Document_ for details of the following data structures processed by move tasks:

* Directory overhead record (DOR)

* File descriptor record (FDR)

* Channel descriptor record (CDR)

* Alias descriptor record (ADR)

* Key indexed file key descriptor record (KDR)

Files or subdirectories with the following names are not automatically processed. Only when specified by directive in the control file are they processed. The names are fixed in the code.

* S$ROLLD

* S$CRASH

* VCATALOG

* S$DIAG

* S$SDTQUE

These names are stored in the data structure SPFMST, which is processed as an exclude list, regardless of the context in which the include/exclude list is processed. SPFMST is searched only if the name is not found in the include/exclude list.

The specific files are excluded because, in general, the data is considered transient, or because processing the file may result in system failure because of the requirement for exclusive access to the file. For instance, if the system swap file is being copied and the move task needs more memory, a deadlock results -- the Get Memory SVC requires that the move task be swapped out.

The FDR of a directory file resides in the parent directory. The VCATALOG FDR resides within the VCATALOG directory file. The VCATALOG FDR is never processed in the first-level directory, since processing it initiates an infinite loop. When the move task encounters a directory FDR, it stops processing the current directory and starts processing the directory associated with the FDR. In the case of a first level VCATALOG FDR, the associated directory is the one currently being processed.

Under no circumstances is a temporary file copied, even if there is a directive in the control file. Information in the FDR determines whether or not the file is temporary.

Directory files are not copied when a hierarchical structure is copied. If the destination directory does not exist, the hierarchy is duplicated by creating an empty directory of the same size. FDRs in this directory are rewritten as files and/or subdirectories are copied. When space is allocated for a file or subdirectory, the FDR is built in the new directory file. With this scheme, when a system crash occurs, no more than one FDR that does not have a corresponding file is left in the new directory.

9.2.1.1 Structure of Tasks.

Each move task uses routines in the .SCI990.S$OBJECT directory. That library appears in each of the link streams.

Each task consists of one task segment. The move tasks include the following categories of modules:

* CD - Highest-level control routine, common to all move tasks. This module contains the transfer vector, and must be linked first in each task.

* Common modules - Routines that are identical regardless of the function being performed

* xxNAME modules - Modules of common design and/or purpose. xx is replaced with the two-character task name to form the name of the module to be linked into that task. For example, each task has an xxDOR module, and VBDOR is the name of the module linked in the verify backup (VB) task. Each xxDOR module processes a directory overhead record as required by the function of the move task xx.

* Modules unique to the task

* Modules from the UTCOMN directory

9.2.1.2 I/O.

S$ routines are used to access PARMS on the bid statement and to write messages to the listing file.

The backup tasks, BD, RD, and VB, allow the use of multiple volumes when the sequential access name specifies a device. The device can be either a disk or a magnetic tape. If it is a disk, BD allocates an image file that is extended, as required, up to the available space on the disk. Blocked records are written to this image file. A minimum block size of 2304 bytes is used; 2304 is the smallest integer of which 256 (sector size of DS300, DS80,WD-800, and CD1400) and 288 (sector size of DS200, DS50, DS10 and FD1000) are factors. Choosing this block size ensures that either sector size is blocked efficiently. The block size used is determined by the amount of free memory available for buffer space. BLKSIZ is set to the highest multiple of 2304 that is less than or equal to the amount of available memory.

Blocking.

When the blocking option is specified in a BD operation, physical records are blocked before they are written to the destination file or device. This provides more efficient use of space on the destination device.

When blocking is specified, records are packed into buffers BLKSIZ bytes in length. (If no block size is specified, a default value of 9600 bytes is used. When BLKSIZ is 9600 bytes, approximately 10 percent of a magnetic tape is used for interrecord gaps.)

Each blocked record starts on a word boundary, and is preceded by a two-byte count. This count is the byte count of the record, except in the following special cases:

* A count of zero marks the end of useful data in a (short) block.
* A count of >FFFF marks the EOF.
* A count of >FFFE marks block of data read using direct I/O, rather than record by record.

Records may span blocks.

The first physical record of the sequential file is a header record. It is a 160-byte record that contains the following information:

Byte	Data
0-5	ASCII text **HDR*. Identifies a header record to RD & VB.
6-13	Binary creation data. Same format as returned by the Get Time and Date SVC.
14-15	Binary volume number.
16-17	Binary block size BD used.
18-19	Binary sector size of source directory
20-21	Flags Bit 0=1 => made with a system that has new EOT handling. Bit 1=1 => made by BDD.
22-63	ASCII time and date of backup.
64-79	ASCII text identifying the volume number.
80-95	ASCII text identifying the sector size.
96-159	ASCII text identifying the source directory pathname.
160-161	Binary fast flag. = 0 if NOFAST was in effect for the backup. -1 if FAST was in effect.

Header Placement and Volume Numbers.

The task's general buffer area is used for blocking buffer memory. As a rule, GETMEM manages this space, but in the case of BD, the values of BUFFER and MEMORY are altered to protect the blocking buffer. (The variables BUFFER and MEMORY are discussed later in this section.)

Double Buffering.

In order to improve performance, BD and RD double buffer when there is enough memory available to allocate two buffers of the specified block size. The read and write routines wait for the preceding read/write if preparation of the second buffer is complete before the read/write of the previous buffer is complete.

Double buffering is not attempted unless the block option is specified.

The VB task does not double buffer I/O, but two buffers are used to load equivalent source file and destination file blocks into memory for the compare. Memory must be available for both buffers or the VB task terminates.

Direct I/O.

For performance reasons, some move tasks use direct disk I/O. Direct disk I/O is discussed in detail in the Supervisor Call (SVC) Reference Manual.

VC does no direct disk I/O.

BD reads the source directory using direct disk I/O, when the destination is a disk device. A destination with an access name of the format DSmn, where m and n are digits, is assumed to be a disk device.

RD uses direct disk I/O to write to the destination when the sequential medium is a disk device (DSmn).

NOTE

> BD and RD code contains comments that refer to direct I/O on the sequential medium. In DNOS, this I/O is blocked file I/O, using very large buffers.

VB reads the file (hierarchical structure) using direct disk I/O, when the sequential medium is a disk device.

CD uses direct disk I/O when it does not jeopardize the integrity of the destination file. Direct disk I/O is not used in the following cases:

* Copying a program file to an existing program file. Both program files are handled, not as one stream of data, but as a collection of tasks, procedures and overlays.

* Copying a program file that contains unused space because tasks, procedures, and/or overlays have been deleted from the program file. One of the functions of CD is to compress program files, and direct disk I/O does not accomplish this goal.

Files that are not program files are subjected to additional tests to determine whether or not to use direct disk I/O. Stated in general terms, using direct disk I/O must not result in a file with an incorrect internal structure. Direct disk I/O is not used if one or more of the following conditions would be created in the output file:

* Unused space where a physical record should start

* A physical record that violates the rule that any physical record spanning allocatable disk units (ADUs) must begin on an ADU boundary. (A physical record that begins in the middle of an ADU must not extend beyond the end of that ADU.)

Figure 9-1 describes the logic used to determine whether or not the current combination of physical record size, sector size, and ADU size allows the use of direct disk I/O. The process is shown in three parts. The first part covers all the special cases that do not require examination of the relationships between the physical record size and the ADU sizes. The second part eliminates from consideration any source file that contains wasted space. Wasted space is caused by inefficient physical record size definition when the file is created. Some cases that could be processed using direct disk I/O are eliminated in this part, because the code is designed never to copy wasted space. The third part of the code eliminates cases in which the output file created using direct disk I/O violates the ADU spanning rule.

The code corresponding to the logic shown in Figure 9-1 is in module DRCTIO. The three parts are not delineated in DRCTIO; they are shown here to highlight the logic. Comments in the code refer to MULTIPLES and SUBMULTIPLES of ADUs and physical records. This is implemented in the code and shown in Figure 9-1 as a test for integer value. The instructions used to make the determination are a divide, followed by a check of register four (the remainder) for zero.

Known: The input file is internally correct.

Define: X=(physical record size/sector size)
X is the number of sectors each physical record occupies in the source file.

ADUDEST = the number of sectors per ADU on the destination device.

ADUSOURCE = the number of sectors per ADU on the source device.

DDIO is an abbreviation for direct disk I/O

```
Part I:  Special Cases

      Are the sector sizes the same on
          source and destination?     ----no--> DO NOT USE DDIO
                       |
                      yes
                       |
                       V
       Are the ADU sizes the same on
          source and destination?    ----yes--> USE DDIO
                       |
                       no
                       |
                       V
       Is the physical record length
             less than a sector?     ----yes-->  USE DDIO -- In this
                       |                 case, every sector contains
                       V                 data and no physical record
                   Part II               spans a sector (or ADU)
                                         boundary.
```

Figure 9-1 CD Logic - Whether to Use Direct Disk I/O (Sheet 1 of 2)

```
   Part II:  Eliminate cases in which there is wasted space in the
             source file.  We have already determined that the
             ADU sizes are not the same on the source and destination.

                        Is X an integer? --no--> DO NOT USE DDIO --
                               |           Part of the last sector in
                              yes          each physical record is
                               |           wasted.
                               |
                               V
       Each physical record is an exact number of sectors
    +----- yes ---------  X > ADUSOURCE?  --------- no ----+
    |             Physical records span ADUs?              |
    |                                                      |
    Is                                                    Is
X / ADUSOURCE                                      ADUSOURCE / X
 an integer? --- yes ->--------+--------<- yes -----  an integer?
    |                          |                          |
    no                         |                          no
    |                          |                          |
    V                          |                          V
DO NOT USE DDIO                |                   DO NOT USE DDIO
See Example A                  |                   See Example B
                               V
                            Part III
```

```
   Part III:  Eliminate cases that create an output file with an
              incorrect internal structure.  It is known that the
              source file has no wasted space.

                               |
       Will physical records span ADUs on output device?
     +-----<--- yes --- X > ADUDEST ? ------- no --->----+
     |                                                   |
     |                                                   |
Is X / ADUDEST                                    Is ADUDEST / X
  an integer? ------ yes -->--+-----<--- yes -------- an integer?
     |                        |                       |
     no                       |                       no
     |                        |                       |
     V                        |                       V
DO NOT USE DDIO               |                  DO NOT USE DDIO
 See Example C                |                   See Example D
                              V
                           USE DDIO
```

Figure 9-1 CD Logic--Whether to Use Direct Disk I/O (Sheet 2 of 2)

The following examples illustrate cases in which CD does not use direct disk I/O. The sector boundaries are marked by ¦, ADU boundaries by a, the physical records by a string of digits and wasted space by /. The conflicts that would be created by using direct disk I/O are shown.

```
Example A
            X = 4, ADUSOURCE = 3, ADUDEST = 1 and 9

   Source:
   ¦11111¦11111¦11111¦11111¦/////¦/////¦22222¦22222¦22222¦...
   a                 a                 a

   Destination (ADUDEST = 1)
   ¦11111¦11111¦11111¦11111¦/////¦/////¦22222¦22222¦22222¦...
   a     a     a     a     a^    a     a     a     a     a
       Physical record 2    ¦
   expected to start here --+


   Destination (ADUDEST = 9)
   ¦11111¦11111¦11111¦11111¦/////¦/////¦22222¦22222¦22222¦...
   a                        ^                            a
        Physical record 2   ¦                            ¦
   expected to start here --+     Improper ADU spanning -+

Example B
            X = 2, ADUSOURCE = 3, ADUDEST = 1 and 9

   Source:
   ¦11111¦11111¦/////¦22222¦22222¦/////¦33333¦33333¦/////¦
   a                 a                 a                 a

   Destination (ADUDEST = 1)
   ¦11111¦11111¦/////¦22222¦22222¦/////¦33333¦33333¦/////¦...
   a     a     a^    a     a     a     a     a     a     a
                ¦   Physical record 2
                +-- expected to start here


   Destination (ADUDEST = 9)
   ¦11111¦11111¦/////¦22222¦22222¦/////¦33333¦33333¦/////¦...
   a            ^                                        a
                ¦   Physical record 2
                +-- expected to start here
```

Example C

```
                  X = 4, ADUSOURCE = 1, ADUDEST = 3

   Source:
   |11111|11111|11111|11111|22222|22222|22222|22222|33333|...
   a     a     a     a     a     a     a     a     a     a

   Destination (ADUDEST = 3)
   |11111|11111|11111|11111|22222|22222|22222|22222|33333|...
   a                 a                 a                 a
                                       ^                 ^
                                       |                 |
                      Improper ADU spans --+              --+
```

Example D

```
                  X = 4, ADUSOURCE = 1, ADUDEST = 9

   Source:
   |11111|11111|11111|11111|22222|22222|22222|22222|33333|...
   a     a     a     a     a     a     a     a     a     a

   Destination (ADUDEST = 9)
   |11111|11111|11111|11111|22222|22222|22222|22222|33333|...
   a                                                     a
                                                         ^
                                                         |
                                     Improper ADU span  --+
```

9.2.1.3 Traversing a Hierarchy.

Directories are hierarchical structures. Move tasks use a stacking technique in traversing the tree structure represented by the source directory.

FDRs in a directory identify either a user data file or another directory. When an FDR that points to a directory is processed, the following information is stacked before the new directory is opened:

* LUNO for the current directory

* Record number of the current directory entry

FDRs are then processed from the new directory file. When the end of the directory is encountered, the directory level is decremented and the LUNO/record number is unstacked. Figure 9-2 shows an example directory structure and the traversing order that results.

DIRECTORY STRUCTURE:

```
                          A
                 +--------+------+-----+
                 |        |      |     |
                .Y        |     .M    .N
                         .X
                   +------+------+
                   |      |      |
                  .B     .C      |
                                .D
                           +-----+-----+
                           |     |     |
                          .U    .V    .W
```

DIRECTORY FILES:

```
A                +--->A.X              +->A.X.D
+-------------+  |    +-------------+  |  +-------------+
|      Y      |  |    |      B      |  |  |      U      |
|-------------|  |    |-------------|  |  |-------------|
|      X      |--+    |      C      |  |  |      V      |
|-------------|       |-------------|  |  |-------------|
|      M      |       |      D      |--+  |      W      |
|-------------|       +-------------+     |-------------|
|      N      |
+-------------+
```

TRAVERSING ORDER:

```
       A.Y
       A.X.B
       A.X.C
       A.X.D.U
       A.X.D.V
       A.X.D.W
       A.M
       A.N
```

Figure 9-2 Example of Traversing a Hierarchy

The stacking area is large enough to support a stacking depth of 15 levels. In the 62-byte stack, the first word is a pointer to free space in the stack. Each four-byte entry contains the LUNO in the rightmost byte of the first word and the record number in the second word.

The VC task maintains two directory stacks -- one for each of the directories being traversed.

9.2.1.4 Control File.

A control file is optional. If no control file is specified, the tasks flow as though there is a control file with only an end directive.

While an 80-byte record length in the control file is not specifically enforced in move tasks (record length is not checked and no error message is generated), only the first 80 bytes are read. Control files can have comments. A ! character causes the rest of the record to be ignored.

9.2.1.5 Error Processing.

The philosophy of error processing in move tasks is that if an I/O error occurs, a message is written to the output listing file and the task continues. Register zero is used throughout to pass along error codes and indicators.

Move tasks end abnormally when the following errors occur:

* Task is unable to get memory for buffers. (This condition does not cause abnormal termination when attempting to get memory for double buffering.)

* System data cannot be read or contains inconsistent or erroneous information. For example, if a directory file appears to extend beyond the end-of-medium, the move task terminates.

The inability to access a file does not cause abnormal termination. The condition is reported, and processing continues with the next file or directory in the traversing order.

A completion code is reported to the user through S$TERM.

9.2.1.6 Volume Numbers in Backups.

When direct disk is selected for the output of a BD, a control file cannot specify the destination in any move directive. This condition is not flagged as an error by the utility, but it produces a backup that does not have all the information in it. When done as required each volume of the backup increments the volume number in the header by one (in module EOTCHK) and retains the same date as the first volume header.

When sequential file is selected as the output of a BD, and no move directive specifies a destination, there is only one header that states volume 1. Any move directive that states a destination must state a new file or it will overwrite the stated file with the new directory. Each new file will have a header that specifies volume 1 and has a new date in it.

When tape is selected as the output of a BD and no move directive specifies the destination tape, a header is written at the beginning of the first tape which specifies wolume 1 and gives the time and date of the backup. Each time an EOT mark is encountered the volume number is incremented by one (in module EOTCHK) and a header written using the same time and date as the original header.

When a control file contains move directives that specify the destination tape, the user is prompted so he can mount a new tape if he desires. After the user's response, a header is written with the same volume number as the previous header and a new time and date.

9.2.1.7 Volume Number Checking by RD & VB.

The method of handling sequential file and direct disk flow directly from the backup handling, the reason being that there is no variation from previous releases' handling of those cases. The method of handling tape, however, is complex because it was written for DX10 to handle any (reasonable) backups made and the code was copied verbatim for DNOS. Module RDHDR contains all the code to handle these cases, except that EOTCHK increments the volume number when an EOT (old backup) or trailer label (backup made by DX10 3.6 or DNOS 1.2) is encountered. There are basically 2 formats that need to be handled:

1. Each MOVE specifying the destination tape increments the volume number and uses the same date as the first header

2. Each such move specifies the same volume number as the preceding header, but gives a new time and date.

In addition, two other conditions are handled:

1. Some releases of the software wrote an EOF mark prior to writing the header produced by a MOVE directive.

2. A user could specify the same tape as output on several move directives. It was desirable for such a tape to be restorable using a control file that did not specify the tape in exactly the same places. This case was also related to correctly restoring such tapes when double buffering was used.

WARNING

These existing schemes in their variety have exhaused the set of schemes which give RD the ability to detect out-of-sequence mounts of tapes, and still be compatible with all schemes. Any new scheme will require a conscious decision to drop compatability with some previous scheme.

9.2.2 Data Structures and Variables.

Move tasks use the following data structures and variables:

* BLKFLG. Two-byte structure that contains flags to indicate whether or not records are blocked. The first byte is for the source file and the second byte is for the destination file. A value of 0 indicates that the associated file is not blocked. A value of >FF indicates that the associated file is blocked.

* CMDFLG. Flag to indicate whether or not a disk is specified as the sequential medium for a backup function (BD, RD, or VB). A value of -1 indicates that the sequential medium is a disk. A value of 0 indicates that it is not a disk.

* DBLFLG. Flag to indicate whether or not I/O is being double buffered. This flag is always set when magnetic tape is the sequential medium.

* BUFFER. Address of the beginning of free memory.

* MEMORY. Amount of free memory that has been cumulatively allocated to this task through Get Memory SVCs.

* INEX. Flag to indicate the mode in which the include/exclude list is to be processed, as follows:

  - INEX = 0. Implies that files and/or subdirectories in the list are to be included in the operation

  - INEX = 1. Implies that files and/or subdirectories in the list are to be excluded from the operation.

* HDRPRC. Flag to indicate, on restore or verify backup, whether an attempt has been made, successful or not, to process a header. A value of 0 indicates it has not been attempted, -1 indicates it has. Set by RDHDR, and cleared by VBDOR, RDDOR, and EOTCHK. VBDOR and RDDOR use it to determine, if not set, that RDHDR must be called.

* MVDR. Flag to indicate to MOUNT, RDHDR, and others if the call is a result of a request to move directory, rather than as a result of end of tape or volume. Different processing is performed in various places depending on this flag.

CFDRVR processes the control file and produces a list of names in the source directory that are to be included or excluded from the operation. This list, the data structure INEXST, consists of a pointer to free space, INEXPT, followed by a maximum of 50 entries. Each entry is ten bytes in length, with eight bytes reserved for the name, one byte for a flag that indicates whether or not the name has been processed, and one filler byte for word boundary alignment. Each name in this list is appended to the directory name to form the pathname of a file or subdirectory to be processed.

OPTNS is a table used in processing the OPTIONS directive in the control file. Options have two states, for example, ALIAS or NOALIAS. All text strings that represent a state of the option are in the xxTEXT module, beginning at label OPTAL. Each six-byte entry in the OPTNS table contains the following information for the particular option:

1. Pointer to the text for -1 value of option

2. Pointer to the text for 0 value of option

3. The option flag that indicates the current state of this option, as follows:

   a. -1. Text matches the -1 text.

   b. 0. Text matches the 0 text.

   c. 1. The user did not specify the option.

The end of the table is marked with a value of 0 in the first word of what would be the next entry (the pointer to the text for the -1 value).

Each entry has a label defined for the flag. The label is a six-character string consisting of a three-character approximation of the option name (for example ALI for ALIAS/NOALIAS and DAT for DATE/NODATE), followed by FLG. Processors use the flag as an external reference to determine the state of the option.

The initial state of each option is the default. The defaults options for move tasks are shown in Table 9-1.

9.2.3 Invoking Move Tasks.

The move tasks are bid by SCI with the following PARMS list:

PARM	Data
1	Input directory (BD, CD) Input sequential (RD, VB) Master (VC)
2	Copy directory (RD,CD,VC,VB) Output sequential (BD)
3	Listing access name
4	Control access name
5	Direct disk sequential backup pathname (BD,RD,VB)
6	Options
7	Blocking factor (BD)
8-12	Date YR,MO,DAY,HR,MIN
13	PRL for SPRL/RPRL (CD,VC)
14	Sequential pathname type 3 for CD, VC For BD, RD, VB: 2 => sequential is direct disk 3 => sequential not direct disk

With the exception of the last parameter, all parameters are discussed in detail in the section of the _DNOS System Command Interpreter (SCI) Reference Manual_ that discusses the Copy Directory (CD) command.

If no options information is specified on the bid statement, the values shown in Table 9-1 are used.

Table 9-1 Default Options for Move Tasks

Task	Defaults
CD	ADD, ALIASES, NODATE, NOSPRL, NORPRL
BD	ALIASES, NODATE, NOBLOCK, NOREWIND, NOUNLOAD
RD	ADD, ALIASES, NODATE, NOREWIND, NOUNLOAD
VB	ALIASES, NODATE, NOREWIND, NOUNLOAD
VC	ALIASES, NODATE, NOSPRL, NORPRL

9.2.4 Internationalization.

Internationalization of move tasks requires that message texts in the xxTEXT modules be translated. These modules contain all the messages a user sees.

English text is used in the header record on the sequential medium in the BD, VB and RD tasks. The following text appears in the xxDATA modules:

* **HDR*
* VOLUME=
* SECTOR SIZE=
* SOURCE=

9.2.5 Detailed Design.

Many modules of move code are common to each of the five tasks. These common modules can be divided into three functional groups:

* High-level control modules included in the linkstream for each of the five tasks:
  - CD. Module that includes the task transfer vector, does initialization, and calls CFDRVR to process the move
  - CFDRVR. Routine that processes the control file and calls the processor whose address is at label SWDIR.
* Modules that are distinct by task but common in structure and/or purpose:
  - xxDATA. Move task common area and variables unique to the task xx.
  - xxTEXT. Text strings used by the task xx. This includes error messages, report headers, informative messages, and the text of valid options and directives.
  - xxDIR. Major loop of the task xx.
* Service routines that are included in the linkstream for each of the five tasks.

Details of the high-level routines are discussed in the following paragraphs.

9.2.5.1 Routine CD.

CD calls the following routines:

* INITST. Initializes statistics-gathering variables
* INITAL. Does the remaining (general) initialization
* GETPRM. Gains access to the parameters on the .BID statement
* WRTHDR. Writes the header to the listing file
* CFDRVR. Processes the input parameters and calls xxDIR to perform the move task.

If CFDRVR returns an error, CD calls the routine ERROR to process the error and to write a message to the listing file. Registers are set to indicate an irrecoverable error.

Regardless of error conditions, DISPST is called to write statistics to the listing file. (DISPST preserves the registers containing error information.)

CD calls S$TERM to terminate the task, and to report any error conditions to the user.

9.2.5.2 CFDRVR.

This routine processes the control file, if any, and calls the xxDIR routine to do the move. The address of xxDIR is at label SWDIR in xxDATA. The logic of CFDRVR is shown in the following metacode:

```
IF no control file
   THEN Set up defaults to move all the files/subdirectories;
        Call xxDIR;
        Return;
   ELSE Open control file;
        LOOP: Read a control file record;
              Write the record to listing file;
              Determine type of directive;
              Perform preprocessing;
              Call directive processor;
                INCLUDE: If mode is include, add name to INEX;(Note 1)
                EXCLUDE: If mode is exclude, add name to INEX;(Note 1)
                OPTIONS: Set options flags;
                MOV:  Call xxDIR to process the previous MOV; (Note 2)
                      Set new access names;
        UNTIL End of file;
        Call xxDIR to execute the last MOV;
        Return;
ENDIF;
```

Note 1 - Mode is determined by first INCLUDE or EXCLUDE directive. The entire operation is aborted when a change of mode is encountered within the same MOV directive. INCLUDE and EXCLUDE directives are processed by common code.

Note 2 -When a move task is bid, the information passed in the PARMS list is processed as though it is a MOV directive in the control file.

9.2.5.3 xxDIR.

xxDIR is the major loop of the move task. This module opens the source directory and traverses that directory, calling the appropriate routines to perform the function of the task.

The logic in the xxDIR module is as follows:

* Skips temporary files

* Calls for reading the next entry in the current directory

* Calls for stacking, unstacking of directory level

* Calls for processing of the DOR and subsequent CDRs, ADRs, FDRs, and KDRs.

* Processes files and/or subdirectories within the directory according to information in the include/exclude list.

VC and CD use the same DIR routine, CDDIR. At this level, the function of xxDIR is to traverse the source directory. SWDOR and SWFDR, addresses in the xxDATA module, cause the correct process (verify or copy) to be performed by the two tasks.

9.2.5.4 xxDATA.

Since each task links in its own xxDATA module, it is not necessary for all of them to be structured exactly the same way. The xxDATA module includes variables used by routines common to all the move tasks and data unique to the particular move task.

The variable TYPE is used to identify the move task internally. The values are as follows:

1 - Backup directory
2 - Copy directory
3 - Restore directory
4 - Verify backup
5 - Verify copy

The following labels are used in high level routines to load the address for branching to xx routines. In xxDATA modules of a task that does not execute the logic which branches to the routine, the value is set to -1 to resolve the address at link time.

* SWDIR - Entry point of xxDIR

* SWHDR - Entry point of the routine that processes headers on the sequential medium file for the task

* SWDRCT - Entry point of the routine that does direct disk I/O

* SWDOR - Entry point of the routine that processes DORs

* SWFDR - Entry point of the routine that processes FDRs

* SWFILE - Entry point of the routine that processes a file

Some routines are called by name in common modules.

NOTE

In xxDATA modules, some of the entry point labels are defined with a value of -1 when it is certain that the branch to the routine is never executed. The value is assigned to resolve all references to it during the linking process. The value of -1 has no special meaning. Should the code of a move task be altered so that the routine is actually called, the xxDATA module must be altered and the routine included in the link stream.

9.2.5.5 Common Service Routines.

The following routines are used by more than one move task. Register assignments and calling sequences are documented in the code and are not specifically covered here.

ADUBLK.

This routine calculates the number of physical records that can be written in the specified number of ADUs.

NBLKS.

This routine determines the number of blocks in a file for initial file allocation and for copying the file in unblocked relative record mode.

APPEND.

This service routine appends a node to a specified character string to produce a pathname. A period and then characters are appended to the character string until a blank is encountered or until eight characters are appended. The count preceding the pathname is increased to produce the new byte count, including the period.

REMOVE.

This routine removes the rightmost portion of a pathname. The delimiter of what is removed is a period or a left parenthesis. The length of the pathname is adjusted for what is removed. If the entire pathname is removed, .VCATALOG is placed in the buffer. If the caller specified a secondary buffer, the portion of the pathname removed is placed into that buffer.

GETACN.

This routine moves an access name from one buffer to another. The character move is terminated by any illegal pathname character, and the count in the second buffer is set to reflect the length of the access name.

GETACN calls S$MAPS to do synonym substitution.

Leading blanks are deleted.

An error condition is set when the pathname begins with DSmn, where n and m are integers. (DSmn is assumed to be a device name, not a file pathname.)

GETCOM.

GETCOM is a parsing routine that looks for a comma. If the next nonblank character is a comma, no error is reported. Otherwise, the parsing pointer is moved back one character and an error code is returned.

GETDSC.

This routine gets disk information for the specified pathname. If the pathname begins with a period, the system disk PDT is used. Otherwise, the first node of the pathname is assumed to be the volume name, and the system PDT list is searched for the appropriate entry.

Once the PDT is located, a LUNO is assigned to the disk. The following information is stored in a six-byte buffer specified by the calling routine:

* Sector size

* Number of sectors per ADU

* LUNO assigned to the disk

GETEOL.

GETEOL is a parsing routine to locate the EOL marker.

GETTXT.

GETTXT parses a text string. The tables PATHN1 and PATHN are used as ranges of legitimate characters that may appear in the string. The tables are defined in the xxTEXT module. PATHN1 consists of A through Z and PATHN1 consists of 0 through 9 and $. The first character that is not in range is the delimiter. Leading blanks are ignored.

JMPFN.

JMPFN parses a filename. It uses the same character set ranges used by GETTXT.

CKFSTK.

CKFSTK searches the include/exclude stack for filenames that have not been processed. It checks the flag in each entry and writes a message to the listing file if an entry exists that has not been processed.

<u>INCLUD</u>.

INCLUD processes the include directive in the control file. If a change of mode is encountered (that is, if an EXCLUDE directive follows an INCLUDE directive within the same MOV), the error is reported. INCLUD also does error checking for no source directory specified and improper pathnames.

<u>OPTION</u>.

OPTION processes the option directive and sets flags in the options table OPTNS.

<u>SCHFNM</u>.

SCHFNM searches the following two lists for a given filename:

1. Include/exclude list INEX

2. SPFMST, the list of special names that are excluded from processing unless they appear in INEX.

INEX is always searched first.

<u>DATE</u>.

DATE determines whether the file should be excluded from copy because of date considerations. If the date option is not active, or if this is a directory file, nothing is done to inhibit the copy. Otherwise, the update and creation dates are compared with the specified date, and a register is set to -1 if the copy is not to be done because the file has not changed since the date.

<u>DESTIN</u>.

DESTIN processes the destination pathname. If the action is a CD, DESTIN verifies that the pathname is a directory. Otherwise, it closes the old file and opens the new file. If this is a VC operation, GETDSC is called to get disk information.

<u>GETPRM</u>.

This routine processes the PARMS list on the bid statement that invoked the task. The listing file is opened and the appropriate flags are set to reflect the options specified in the PARMS list. If there is not a source and a destination or a control file, an error code is set. (The source and destination can be specified in the control file.)

GETPRM calls S$ routines to access PARMS and to open the listing file.

If the blocking size is invalid, 9600 is used, and processing continues. The blocking size is rounded down, if necessary, to make it even.

The logic to process all elements of the PARMS list is in this module, but, depending on the task, variables are initialized to cause proper flow for the expected PARMS list.

SOURCE.

This routine processes the access name of the source. If it is a file pathname (as opposed to a directory name) variables are set up for calling the copy routine. In either case, the access name is stored in the source buffer and the include/exclude flag is cleared.

ERRINT.

ERRINT reports an internal error and saves variable text for eventual message construction. If an error condition has already been reported, the current error is ignored.

ERRCLR.

ERRCLR resets the error condition previously reported by calling ERRINT.

ERROR.

ERROR is the error processing routine. This routine is called with ERRTYP in register zero. The first byte contains the code for system errors and the second byte contains the code for program-defined errors. A value of >8000 implies an SVC error.

The following actions are taken, depending on the error type:

* SVC error:

  - Gets information from the return code processor call block

  - Sets the error file number

* Internal error:

  - Converts the error code to a message number. (Message numbers are used to access message text in the module xxTEXT.)

  - Gets saved variable text

- Sets the error file number

An error message is created and written to the listing file. The error and the variable text buffer are cleared.

Some internal errors are not reported.

<u>ERRS$</u>.

This routine reports errors returned by S$ routines.

<u>ERRSVC</u>.

ERRSVC examines an SVC call block and reports any errors found.

<u>SETCC</u>.

This routine calculates the value to be used for the $$CC synonym. (It does not set the synonym.) SETCC totals the number of times the error messages in xxTEXT have been written. There is a count byte in the data structure containing the text, and it is incremented each time the message is written. This total for all messages is added to >8000 to produce the condition code that is eventually returned to the user in the synonym $$CC.

<u>GETREC</u>.

This routine gets a specified record in a file being read. The IRB is set up with the record number and the buffer address. READB is called to do the read.

<u>I$O</u>.

I$O structures the SVC block and issues all I/O SVCs for the task. There is an entry point for each SVC listed. Blocks for the following services are formatted:

* Assign a LUNO to the specified access name

* Get the type of pathname in the specified IRB

* Close the specified file

* Close a file without changing the update information

* Close and unload a device

* Delete a file

* Open a file

* Release the specified LUNO

* ASCII operations, including special processing for multivolume considerations:
  - Read a record
  - Write a record
  - Write EOF to specified IRB

<u>OPNFIL</u>.

This routine assigns a LUNO to and opens the file. Program and image files are opened blocked. All other files are opened unblocked.

<u>WRTHDR</u>.

WRTHDR writes the initial header and parameter values to the listing file.

<u>WRTLIN</u>.

WRTLIN writes the contents of a buffer to the listing file as a single line.

<u>WRTLST</u>.

WRTLST uses a variable length parameter list to output user messages via the routines S$WRIT and S$WEOL. It also generates headings and controls paging in the listing file.

<u>INITAL</u>.

This routine does all CD initialization in every move task.

<u>CLRIRB</u>.

CLRIRB clears the contents of an IRB.

<u>MEMMGR</u>.

MEMMGR builds the call block and issues SVCs to get and release memory. It also maintains the variables BUFFER and MEMORY for internal allocation of memory.

<u>STKDIR</u>.

This routine stacks the directory level. The LUNO and record number in the source IRB are saved in the directory stack. A LUNO is assigned to the new access name and it is opened. If there is an error in either the assign or the open, the old IRB is restored and an error code is returned to the caller. If the error is on the open, the LUNO is released.

When there are no errors, the record number in the source IRB is set to 1.

<u>POPSTK</u>.

POPSTK pops the directory level stack. The logic of this routine is shown in the following metacode:

```
Close present source and release LUNO;
IF top of stack
   THEN Return with top of stack signal;
   ELSE Place new LUNO in IRB;
        Place record number into RECNUM;
        Decrement stack level counter;
        Return;
ENDIF;
```

9.3 SUPPORT FOR REMAINING FILE MAINTENANCE UTILITIES

The remaining file maintenance utilities make extensive use of two sets of routines -- those in the directory DSC.O$, and the UTCOMN routines that sort a directory file. This paragraph contains a short description of these support routines.

These tasks also use routines that are in the S$SYSTEM procedure segment. All except CCAF include the procedure segment. CCAF uses the directory VOLOBJ.SCI990.S$SYSTEM as a library in the link stream. This directory contains object of the routines included in the S$SYSTEM procedure segment.

9.3.1 O$ Routines.

The object module O$DTA in the DSC.O$ directory is a data area used by all O$ routines.

The module O$INP contains the following routines:

O$INIT	Initializes tables and sets parameters to obtain PARMS list
O$PARM	Obtains the requested element of the PARMS list and does the requested conversions
O$TERM	Terminates the task through S$STOP. This routine releases the TCA and passes the following message or its local language equivalent:

ERROR IN OUTPUT TERMINATION

The module O$OUT contains the following routines that perform output functions:

O$SOUT	Initializes variables for output functions
O$CHAR	Puts a character string into the line buffer
O$HEX	Puts a hexadecimal number into the line buffer
O$DEC	Puts a decimal number into the line buffer
O$TAB	Puts blanks into the line buffer, to next tab position
O$SPAC	Puts a blank into the line buffer
O$LINE	Terminates the current line, writes it to the listing file, and blank fills the line buffer.

9.3.2 UTSORT.

The UTCOMN module UTSORT contains the following routines used extensively by the list directory and map disk utilities:

SORT	Sorts directory entries. This routine builds two data structures in memory and returns a pointer to the DOR in memory.
NXTENT	Gets the next sorted entry (directory/data file, alias or channel). This routine returns a pointer to the descriptor record for the next entry of the specified type.
CLSSRT	Closes the file opened by SORT and releases the LUNO.
ADDSAT	Computes the number of ADUs used by the entry (that is, the total of primary and secondary allocations.
PMT	Processes move table. Moves an entry in the linked list of sorted entries.

9.3.3 UTSORT Data Structures.

SORT builds two data structures that are available to callers. SDEDOR contains information from the DOR of the file that has just been sorted. SDEMD is a linked list that represents the sorted entries in the directory file. The first record contains information about the directory and space for statistics gathering. The subsequent records contain a filename, the record number for the file, and linking information. Details of these two data structures are included in the data structure pictures section of this manual.

9.4 LIST DIRECTORY (LD)

The LD task formats and writes a summary report of the entries in a directory file. The program is written in assembly language and makes extensive use of UTCOMN and S$SYSTEM routines. It does not use the procedure segment S$SYSTEM.

LD is bid by SCI with the following PARMS list:

PARM	Definition
1	Code. This parameter is not used
2	Input directory name
3	Output access name

LD calls SORT (in UTSORT) to sort the entries in the specified directory file. The header information is extracted from the memory-resident DOR (SDEDOR), formatted into the report header, and written to the listing file. The directory information is processed in three passes:

1. Directories and aliases

2. Files and aliases

3. Channels

Each pass consists of a loop that calls the UTSORT routine NXTENT to determine the next entry of the type being processed in this pass. The information from the returned FDR, ADR or CDR is processed and written to the listing file.

Errors are reported to SCI through the UTCOMN routines UTUERR and UTSERR. SCI reports errors to the user.

9.5 MAP DISK (MD)

MD is an assembly language program that maps the contents of a disk volume, a directory or a file. It produces a report with the level of detail specified by parameters in the PARMS list.

MD is bid by SCI with the PARMS list as follows:

PARM	Definition
1	Code. This parameter is not used.
2	Pathname of directory file
3	Listing access name
4	Short form? (YES/NO)
5	Top level only? (YES/NO)
6	Directory nodes only? (YES/NO)

The report generated by MD is described in the DNOS System Command Interpreter (SCI) Reference Manual in the section that discusses the MD command.

The task segment includes object of the source modules in the DSC.DP.MD.SOURCE directory, UTCOMN routines, the KIFMGR conversion table (KMTAB), and the O$ routines. MD links in the procedure segment S$SYSTEM.

The driver for MD is the module MD. It calls MAPFIL, in the module MDMAPF, to calculate statistics on the top-level directory file and to write that information to the listing file. On subsequent calls to MAPFIL, if the statistics are not written to the listing file (for instance, in processing a data file when the PARMS list specifies directory nodes only), MAPFIL only gathers statistics.

MD calls MAPDIR, in the module MDMAPD, to control the mapping of the directory. MAPDIR calls SORT for the (currently) top-level directory. First directory files and then data files are processed. Channels and aliases are ignored. The processing loop for each pass calls NXTENT for the next sorted entry and calls MAPFIL to calculate statistics and write them to the listing file.

After all files in the current directory are mapped, MAPDIR calls itself recursively to map each directory and subdirectory.

MD uses the UTSORT data structures SDEDOR and SDEMD. If MAPDIR finds one or more directory files, after the files in a directory are processed, MD determines whether or not the memory-resident structure SDEMD needs to be compressed to make space available for subsequent calls to SORT. If so, the data file entries are taken out of the linked list so that SDEMD contains only directory files.

Errors are reported through the UTCOMN routines UTUERR and UTSERR.

9.6 DELETE DIRECTORY (DD)

DD is an assembly language program that deletes all entries in a specified directory, and deletes the directory file itself.

DD is bid by SCI with the following PARMS list:

PARM	Definition
1	Code, this parameter is not used.
2	Pathname of the directory to delete
3	Listing access name

The task segment includes object of the source in module DSC.DP.DD.SOURCE.DD, and UTCOMN routines. DD links in the procedure segment S$SYSTEM.

DD is the entry point where initialization is done, the routine DH is called, and the task is terminated.

DH is a recursive routine that deletes a data file or deletes the entries in a directory file. When it is called by DD, L has a value of >C0 and F points to the pathname of the directory specified on the PARMS list. The logic of DH is shown in the metacode below:

```
DH:  Assign LUNO L to file F;
     IF not a directory
        THEN Go to DIR20;
        ELSE
               Open file
         DIR10:Read next FDR
               IF EOF
                  THEN Close file;
                       Go to DIR20;
                  ELSE IF ADR, CDR, KDR, ACR or unused
                          THEN Go to DIR10;
                          ELSE L=L+1;
                               F'=Pathname F;
                               F=F+FDR pathname field;
                               Call DH;
                               F=F'
                               L=L-1;
                               Go to DIR10;
                       ENDIF;
               ENDIF;
         DIR20:Release file;
               Delete file;
               Log deleted message;
               Return;
END DH;
```

DD uses the UTCOMN subroutine linkage UTPUSH and UTPOP, and DH saves eight registers with each recursive call. These registers contain all information that must be stacked and unstacked in the recursion.

DH makes a special case check for VCATALOG. The volume directory file is never deleted, even though all files for which it contains entries are deleted.

9.7 CCAF

CCAF is a task that performs two file maintenance functions:

* Copies the contents of one file into another file

* Appends the contents of one file to the end of another file

CCAF is bid by SCI with a PARMS list and a CODE value. CODE=1 is for the copy function. CODE=2 is for the append function. The PARMS list is as follows:

PARM	Definition
1	Input access name(s)
2	Output access name
3	ANSI flag (YES/NO)
4	Replace? (YES/NO)
5	Append? (YES/NO)
6	Maximum record length

The input and output access name parameters and the replace option are discussed in the DNOS System Command Interpreter (SCI) Reference Manual in the section that discusses the CC command.

The ANSI flag tells CCAF whether the input file contains ANSI carriage control.

The maximum record length (MRL) parameter is optional. When specified, it governs the amount of information (number of characters per record) written to the output file. MRL also has some effect on the amount of buffer space used by the utility.

The task CCAF consists of the object module CC in the directory VOLOBJ.CC and UTCOMN routines. It uses the library VOLOBJ.SCI990.S$OBJECT for S$SYSTEM routines. (It does not use the shared procedure segment.)

CCAF does initiate I/O and uses double buffering. The buffer size is initially set to the larger of MRL or 80. When MRL is not specified (=0), the buffer size defaults to 512. After the input file is opened and an SVC is issued to get the file characteristics, the size of the read buffer is adjusted, if necessary.

If the fifth parameter is YES, the output access name is checked for the forms DSnn and CRnn (where n is numeric). An error is generated if this is an attempt to append a file to a disk or card reader device. A LUNO is assigned to the output access name, and it is opened. (If the fifth parameter is YES, the file is opened extended.)

Most of the remaining processing is done in routines CPYDAT, PRTDAT, and INITIO.

INITIO initiates a read with one buffer and does a write (with reply) from another buffer. After the write completes, INITIO waits on the read, if necessary.

CPYDAT prepares buffers and calls INITIO.

PRTDAT inserts carriage control, if necessary. PRTDAT is called only if the output access name is a device. The processing is dependent on the format of the input file, as follows:

* Unformatted file - Inserts line feed between records and calls INITIO

* ANSI formatted file - Converts from ANSI to device carriage control codes and calls INITIO

* File that already has device code carriage control - Calls CPYDAT

CCAF terminates when an EOF is encountered and all files specified in the input access name list have been processed. The second and subsequent files are assumed to have the same characteristics as the first file. Buffer sizes and other parameters are not reinitialized. The next input file is opened and copied.

When the input access name list is exhausted, the files are closed and the task terminates through UTUERR.

SECTION 10

USER ID AND ACCESS GROUP MAINTENANCE

10.1 OVERVIEW

The user ID maintenance package and the access group maintenance package are interrelated. User IDs and access groups are maintained in the same file and the tasks share common subroutines.

The user ID maintenance package (AUIDUI) performs the following functions:

* Add a user ID

* Delete a user ID

* Modify a user ID

* List all user IDs

The access group maintenance package (AGTASK) performs the following functions:

* Create an access group

* Delete an access group

* Set a users file creation access group

* Modify an access group

* List all the access groups of which the user is a member

* List the members of all access groups of which the user is a leader

A separate task (MPC) is provided to allow users to modify their passcode.

AUIDUI, MPC, and AGTASK are all written in Pascal and are supported in both interactive and batch modes. These three tasks work together to maintain the file .S$CLF. This file contains all information pertaining to user IDs and access groups.

10.2 STRUCTURE OF THE TASKS

Each task consists of a task segment only. The code is not designed to be sharable. It is replicated each time the program is invoked. These tasks receive input parameters via the standard PASCAL interface to SCI and return errors and messages via standard PASCAL interface to SCI.

These tasks make use of two external functions -- JMHASH, a routine for hashing user IDs into their entries in the .S$CLF file, and PLCRYT, a routine for encrypting passcodes for storage in the .S$CLF file.

10.3 FILES

AGTASK, AUIDUI, and MPC maintain information concerning user IDs and access groups in a system file, .S$CLF. AUIDUI initializes a file in the directory .S$USER each time a user ID is added. This file has the name of the user ID that has been added.

10.3.1 .SCLF.

Management of the .S$CLF file is the primary responsibility of AUIDUI, AGTASK, and MPC. .S$CLF is created when the system is booted the first time. It is a relative record file with 54 byte logical records and 864 byte physical records. This file contains five kinds of records, Verification Record (VFY), File Information Record (FIR), Access Group Record (AGR), User Descriptor Record (UDR), and User Descriptor Overflow record (UDO). The template that describes the five different records is called a Capabilities List file Record (CLR). Each record is described below. Refer to the section of data structure pictures at the back of this manual for a picture of the CLR and its five variants.

10.3.1.1 Verification Record (VFY).

The first record is a header record that contains the name of the file and an indication of the version and release of the operating system that created the file. It also contains a pointer to the first access group record (AGR). The header record is used for verification purposes when the system is booted and by AGTASK to locate the list of AGRs.

10.3.1.2 File Information Records FIR).

Information about a particular user ID is accessed through FIRs. .S$CLF accomodates 53 base FIRs. An unlimited number of continuation FIRs may be linked to each base FIR. (The number of continuation FIRs is unlimited with respect to the UIDTASK code.)

The character string representation of the user ID is passed to JMHASH, which computes a number from 1 through 53. This number is the FIR into which the user ID will be written.

A single FIR contains information for as many as five user IDs. FIRs may be linked to provide as many entries as required to store information for user IDs that hash to the same FIR number.

The FIR contains 10 bytes of information for each user ID. The first eight bytes are used to store the character string representation of the user ID, and the last two bytes contain the .S$CLF record number of the UDR for the ID.

The first word of an FIR indicates whether it has been linked. A value of zero indicates that the user IDs represented in the single record are all that hash to this record number. If the value is nonzero, it is the record number of the continuation record for this FIR.

10.3.1.3 User Descriptor Record (UDR).

A UDR describes the record that represents a given user of the system. It is a 54 byte record that includes the encrypted password, user description, and access group information. There is one UDR for each user in the system.

The UDR can contain access group information for up to 5 access groups. Each access group in the UDR is represented by an access group entry. Each access group entry contains the record number of the access group record, the index into the access group record for this access group, and flags to indicate the users file creation access group and access groups of which the user is the leader.

The first word of the UDR is a pointer to a user descriptor overflow record (UDO) If the user is a member of more than five access groups, a UDO is created. The first word of the UDR contains the record number of the overflow record. If there is no overflow record the first word is zero.

10.3.1.4 User Descriptor Overflow record (UDO).

A UDO describes the record that contains information pertaining to additional access groups of whichs the user is a member. This structure exists only when the user becomes a member of more access groups than will fit in the UDR. This structure has room for up to twelve access groups. Each access group is represented by an access group entry. Each access group entry contains the record number of the access group name record, the index into the access group name record for this access group, and flags. There is no limit to the number of UDOs that may be created for each user ID.

10.3.1.5 Access Group name Record (AGR).

The AGR describes the record that contains the names of access groups. Each access group name record contains space for up to five access group names. Access group names are encrypted before being stored in the AGR. Each access group defined to the system has an entry in an AGR. Each entry consists of the access group name and a reserved word. The first word of the AGR contains the record number of the next AGR. The first word of the last AGR in the chain contains a zero.

10.3.1.6 Structure of the .S$CLF file.

The .S$CLF file is a relative record file and must be expandable. The file is initialized the first time the system is booted. It contains 53 FIRs and 16 AGRs. There are entries in the FIRs for the user IDs SYSTEM and SYSMGR, UDRs for the user IDs SYSTEM and SYSMGR, and an entry in the first AGR for the access group SYSMGR. Additional FIRs, UDOs, FIRs, and AGRs are created as needed. The logical format of .S$CLF is shown in Figure 10-1, and the physical format of .S$CLF is shown in Figure 10-2.

The example shows the following information:

* User ID MICHAEL is a member of access groups MODP and DNOS.

* User ID JIM is a member of access group DNOS.

* User ID DEBBIE is a member of access groups MODP and FILSEC.

DEBBIE is a member of more access groups than will fit in her UDR so a UDO was created.

* Access group MODP has users MICHAEL and DEBBIE as members.

* Access group DNOS has users MICHAEL and JIM as members.

* Access group FILSEC has user DEBBIE as a member.

```
       +-------------------+
       ¦ RECORD 0 OF S$CLF ¦
       +-------------------+-------->+
                                     ¦
                                     ¦   AGR BLOCK
+<-----+---------------+             +---------------+<- PHYSICAL
¦ +<---+ FIR RECORD    ¦             ¦ AGR  RECORD 1 ¦   RECORD
¦ ¦ +<-¦               ¦ +-+-------->¦ DNOS          ¦   BOUNDARY
¦ ¦ ¦  +---------------+ ¦ ¦ +-+---->¦ MODP          ¦
¦ ¦ ¦                    ¦ ¦ ¦ ¦ +-->¦ FILSEC        ¦
¦ ¦ ¦  +---------------+>+ ¦ ¦ ¦ ¦   ¦ EMPTY         ¦
¦ ¦ +->¦ UDR RECORD    ¦   ¦ ¦ ¦ ¦   ¦ EMPTY         ¦
¦ ¦    ¦  JIM          ¦   ¦ ¦ ¦ ¦   +---------------+
¦ ¦    +---------------+   ¦ ¦ ¦ ¦   ¦ AGR  RECORD 2 ¦
¦ !                        ¦ ¦ ¦ ¦   ¦ EMPTY         ¦
¦ +-->+----------------+-->+ ¦ ¦ ¦   ¦ EMPTY         ¦
¦     ¦ UDR RECORD     ¦---->+ ¦ ¦   ¦ EMPTY         ¦
¦     ¦  MICHAEL       ¦       ¦ ¦   ¦ EMPTY         ¦
¦     +----------------+       ¦ ¦   ¦ EMPTY         ¦
¦                              ¦ ¦   +---------------+
+---->+----------------+------>+ ¦   /               /
      ¦ UDR RECORD     ¦         ¦   /               /
+<----¦   DEBBIE       ¦         ¦   +---------------+-->+
¦     +----------------+         ¦   ¦ AGR  RECORD 16¦   ¦
¦                                ¦   ¦               ¦   ¦
¦     +----------------+         ¦   ¦               ¦   ¦
+---->¦ UDO RECORD     ¦-------->+   ¦               ¦   ¦
      ¦                ¦             ¦               ¦   ¦
      +--------------- +             ¦               ¦   ¦
                                     +---------------+   ¦
                                                         ¦
                                                         ¦
                                                         ¦
                                                     POINTER
                                                     TO NEXT
                                                     AGR BLOCK
                                                     IF NEEDED
```

Figure 10-1 Logical Organization of S$CLF

```
RECORD                     CONTENTS

        +-----------------------------------------------+
0       ¦ VERIFICATION RECORD                           ¦
        +-----------------------------------------------+
1       ¦ First  FIR                                    ¦
        +-----------------------------------------------+
        /                                               /
        /                                               /
        /                                               /
        +-----------------------------------------------+
53      ¦ FIR number 53                                 ¦
        +-----------------------------------------------+
        /                                               /
        / UDRs, UDOs, FIRs                              /
        /                                               /
        +-----------------------------------------------+
64      ¦ First AGR                                     ¦
        +-----------------------------------------------+
        /                                               /
        +-----------------------------------------------+
79      ¦ Last AGR                                      ¦
        +-----------------------------------------------+
        /                                               /
        / UDRs, UDOs, AGRs, FIRs                        /
        /                                               /
        +-----------------------------------------------+
```

Figure 10-2 Physical Organization of .S$CLF

The first bit in the second word of each record is used to indicate whether the record is in use. When the file is initialized this flag is set to indicate the record is available. Each time a new FIR, UDR, or UDO is needed, the file is searched for the first available record. When one is found, the used bit is reset to indicate the record is in use. Each time a UDR, UDO, or an FIR other than one of the original 53 is no longer needed, the used bit is set. When new AGRs are needed, they are always allocated in contiguous blocks of 16 records on physical record boundaries. This causes entire physical records to be dedicated to AGRs. This scheme minimizes the number of disk accesses necessary to read the list of access groups needed for security access checking.

10.3.2 Synonym and Logical Name File.

User ID maintenance initializes this file when a user ID is added.

It is set up to appear to Name Manager as a synonym and logical name segment with no synonyms or logical names defined.

UIAUI expects the value of the synonym SESYN to be the pathname of the synonym and logical name file. This file must be a relative record file that contains only one record.

See the DNOS System Design Document for details of data structures used by Name Manager.

10.4 FLOW OF CONTROL OF AUIDUI

The following paragraphs describe the major phases of AUIDUI--invoking the task, initialization, major routines, and termination.

10.4.1 Invoking AUIDUI.

AUIDUI is invoked by command procedures under SCI. The task is bid by SCI990 with a PARMS list.

The first three PARMS are the same for each of the user ID commands:

```
PARMS=(stack parameter, heap parameter, function code...)
```

Stack and heap parameters of 6000 and 1000, respectively, are large enough for the program as shipped by Texas Instruments Incorporated.

The remainder of the PARMS list depends on what function is being requested. Listed below is the information expected by each of the functions in elements 4 through n of the PARMS list.

Function Code	Process/ SCI Command	PARMS List (past function code)
1	Assign User ID (AUI)	New user ID, new passcode, user privilege level, user description
2	Delete User ID (DUI)	User ID
3	Modify User ID (MUI)	User ID, new passcode, user privilege level
4	List User IDs (LUI)	Output access name

In addition to the BID statement, user ID maintenance requires that the synonym SECLF be the pathname of the system file .S$CLF. When a user ID is to be added, an additional synonym must be set prior to bidding the task. The synonym SESYN is expected to be the pathname of a synonym and logical name file. Command procedures that invoke AUIDUI should set this synonym and create the directory .S$USER.<user ID>.

10.4.2 Initialization.

The entry point for user ID maintenance is the Pascal procedure UIMAIN. The function code is isolated and an SVC issued to establish the user ID of the job that invoked the task. The privilege level of the invoking user is checked to verify that the requested action is allowed.

At the end of the routine, a case statement based on the value of the function code transfers control to the routine that does the processing.

10.4.3 Major Routines.

The following paragraphs describes the major routines in AUIDUI.

10.4.3.1 Add User ID - UIAUI.

The routine UIAUI in the procedure UIMAIN processes the addition of a user ID to the .S$CLF file and initializes the synonym and logical name file.

If the user ID is already in the file, an error is returned through UTPUER.

The process includes building an FIR entry and an associated UDR. They are written to .S$CLF.

10.4.3.2 Delete User ID - UIDUI.

Deletion of user IDs is processed by the routine UIDUI in the procedure UIMAIN. This processing requires that the user ID be cleared from the FIR and that the UDR space be freed. If the deletion empties an FIR continuation record, the record is marked unused and the record is unlinked from the previous, and where appropriate, next record in the chain.

10.4.3.3 Modify User ID - UIMUI.

Modification of user ID attributes (passcode, privilege level and/or description) is processed by UIMUI in the procedure UIMAIN.

The UDR for the specified user ID is read, altered as specified, and rewritten to the .S$CLF file. A null value for passcode, privilege level, or description is processed as a no change request for that attribute.

10.4.3.4 List User IDs - UILUI.

The routine UILUI in the procedure UIMAIN processes the request for listing user IDs. All nonempty FIRs are read to obtain active user IDs. For each ID, user description and privilege level are read from the associated UDR. This information is used to construct an element in a linked list. When all IDs have been read, the linked list is sorted by user ID, in alphabetic order. The list, along with the time and date, are written to the specified listing file.

10.4.4 Termination.

Termination of user ID maintenance is through R$TERM when there are no errors to report, and through S$TERM when an exit is taken to report errors through UTPUER.

10.5 FLOW OF CONTROL OF MPC

MPC is invoked by command procedures under SCI. The task is bid by SCI990 with a PARMS list. The first two parameters are stack and heap sizes, the third parm is expected to be the old passcode, the fourth is expected to be the new passcode.

The old passcode is encrypted and compared with the passcode stored in the .S$CLF file. If the passcode does not match, the processing is aborted and an error is returned to the user. If the passcode matches, the new passcode is encrypted and placed in the users UDR. The UDR is then rewritten to the file .S$CLF.

10.6 FLOW OF CONTROL OF AGTASK

The following paragraphs describe the major phases of AGTASK -- invoking the task, initialization, major routines, and termination.

10.6.1 Invoking AGTASK.

AGTASK is invoked by command procedures under SCI. The task is bid by SCI990 with a PARMS list.

The first three parms are the same for each of the user ID commands:

```
    PARMS=(stack parameter, heap parameter, function code, ...)
```

Stack and heap parameters of 6000, and 1000, respectively, are large enough for the program as shipped by Texas Instruments Incorporated.

The remainder of the parms list depends on what function is being requested. Listed below is the information expected by each of the functions in elements 4 through n of the PARMS list.

Function Code	Process/ SCI Command	PARMS List (after function code)
0	Create Access Group (CAG)	Access group, users to add
1	List Access Group (LAG)	NULL, NULL, passcode, output access name
2	Set Creation Access Group (SCAG)	Access group, NULL, passcode
3	Delete Access Group (DAG)	Access group, NULL, passcode
4	List Access Group Members (LAGM)	Access group, NULL, passcode output
5	Modify Access Group (MAG)	Access group, users to delete, passcode
6	Modify Access Group (MAG)	Access group, users to add, passcode
7	Modify Access Group (MAG)	Access group, NULL, passcode new leader

10.6.2 Initialization.

The entry point for access group maintenance is the Pascal procedure AGTASK. A retrieve system data SVC is issued to verify the system is DNOS 1.2 or later. The function code is isolated and many checks are performed to verify that the request is valid. For every operation except CAG the users passcode is verified against the passcode for the job issuing the command. For every function except CAG, SCAG, and LAG the user must be the leader of the access group or a member of the SYSMGR access group. For every function except CAG the access group must exist. After initial verification the function code is used to transfer control to the appropriate routine.

10.6.3 Major Routines.

The following paragraphs describe the major routines in AGTASK.

10.6.3.1 Add list of users to access group (AGADLU).

AGADLU is a routine to add a list of users to an access group. It gets the list of users as a PARM from SCI. User IDs are split from the list and processed one at a time. Each user ID is checked to verify it exists and the user is not already a member of the access group. If the user is a member of the SYSMGR access group he cannot become a member of any other access groups. Similarly, if the user is a member of other access groups he cannot become a member of SYSMGR. After the user ID is checked, an entry for this access group is added to the user's UDR or UDO. An informative message is written to the TLF to indicate which user IDs have been added, which have not, and why.

10.6.3.2 Create access group (AGCAG).

AGCAG is a routine to create an access group. The access group is checked to verify that it does not already exist. If it does not already exist, the access group name is encrypted and an entry is made in the next available AGR for this access group. If no entries are available in existing AGRs, a block of 16 AGRs are created, initialized, and the encrypted access group name is added to the first new AGR. The user who issued the request to create the access group is added to the access group as the leader by adding an entry is his UDR or UDO. AGADLU is then called to add the list of users specified when the CAG command was issued.

10.6.3.3 Change access group leader (AGCHGL).

The routine AGCHGL is a routine to process requests to change leadership of an access group. The user ID of the new leader is checked to insure it is a valid user ID. If the requestor is a member of the SYSMGR access group, a search of every user ID is performed to locate the old leader of the access group. The new leader is added to the access group if he is not already a member. In case of a crash it is better to have two leaders instead of none so the leader flag in the new leader's UDR or UDO is set before the leader flag in the old leader's UDR or UDO is reset.

10.6.3.4 Delete access group (AGDAG).

The routine AGDAG processes the deletion of an access group. The access group cannot be deleted if it has members. Every UDR and UDO is checked to verify that any member of the specified access group is also the leader. The access groups PUBLIC and SYSMGR cannot be deleted. After these checks the entry for this access group in the leader's UDR or UDO is cleared. The AGR which contains the entry for this access group is located and the entry is cleared.

10.6.3.5 Delete users from access group (AGDEL).

The routine AGDEL deletes a list of users from an access group. It gets the list of users as a PARM from SCI. User IDs are split from the list and processed one at a time. Each user ID is checked to insure that it exists. If it exists the UDR and UDOs are checked to insure the user is a member and is not the leader. After these checks, the entry for this access group in the users UDR or UDO is cleared. An informative message is written to the TLF indicating which users were deleted, which were not, and why.

10.6.3.6 List access groups (AGLAG).

The routine AGLAG processes requests to list all access groups of which the requestor is a member. If the requestor is a member of SYSMGR all access groups are listed by reading all AGRs, decrypting the access group name, and inserting it into a linked list in sorted order. If the requestor is not a member of SYSMGR his UDR is located. For each entry in the users UDR and UDOs the access group name is read, decrypted, and inserted into a linked list in sorted order. When all access groups have been located, the list is output to the specified listing file along with the date, time, an indication of those access groups of which the user is the leader, and an indication of the users file creation access group.

10.6.3.7 List access group members (AGLAGM).

The routine AGLAGM process requests to list the user IDs of all members of the specified access group. Every UDR and UDO is read in search of entries corresponding to the specified access group. Each time a match is found, the user ID is inserted into a linked list in sorted order. When all UDRs and UDOs have been searched, the list is output to the specified listing file along with the date, time, and an indication of which user ID is leader of the specified access group.

10.6.3.8 Set file creation access group (AGSCAG).

The routine AGSCAG processes requests to change a user's file creation access group. The users UDRs and UDOs are searched. The user's previous file creation access group flag is reset if one is found. If the specified file creation access group is not PUBLIC, the file creation access group flag in the entry corresponding to the specified access group is reset. An error is returned if the user is not a member of the specified access group.

10.6.4 Termination.

Termination of AGTASK is through R$TERM if there are no errors to report. When there are SVC errors to report termination is through UTPSER. When there are utility errors to report termination is through UTPUER.

SECTION 11

TELEPRINTER DEVICE UTILITIES

11.1 OVERVIEW

The set of tasks provided as teleprinter device utilities provide control of the entire range of supported hardcopy terminal products. The utilities are accessable to a terminal user through use of SCI either interactively or via batch stream execution as shown in Figure 11-1. The user may be using the terminal to which the commands are addressed or another terminal, to the extent that each makes sense.

All tasks defined here use standard conventions for interfacing between SCI and the control task. These are

1. No synonyms are set by the task nor expected to be set prior to bidding the task except as specifically noted.

2. All responses to prompts are passed to the bid task as PARMS. The order of the PARMS is the same as the order of the prompts. No PARMS which are not prompt responses are passed.

3. Most tasks have various "tuning parameters". If there are such parameters, they are listed as PARMS immediately following the prompt responses. Their order is as noted in this section.

4. The CODE value for the .BID is set to zero except for those cases specifically noted in the rest of this section.

5. The tasks are installed in S$UTIL.

6. The task name for each task is noted as part of the description.

```
                      !----------!
                      !  Procs * !
                      !          !
                      !          !
                      !----------!     !----------!
                            !          !          !
                            !         >!  User    !
                            !        //!          !
                            V       // !----------!
      !----------!    !----------! //
      ! Logical  !--->!  S       !//
      ! Name/Syn !    !     C    !<
      ! Segment  !<---!        I !\\
      !----------!    !----------! \\
              !                     \\ !----------!
         !    !                      \\!  Batch   !
         !    !                       >!  Stream  !
         !    V                        !          !
      !-----------!                    !----------!
      ! TPD       !                    !  Batch   !
      ! Utilities*!                    !  Listing !
      !           !                    !          !
      !-----------!                    !----------!
               !
          !    !
          !    !           * Covered by this section
          !    V
       !----------!
       !  D       !
       !     S    !
       !        R !
       !----------!
```

Figure 11-1 Interfaces Between SCI and Control Tasks

11.2 COMMANDS

The commands supported by the teleprinter device utilities utilize standard DNOS SCI interfaces. Terminology used in these commands corresponds to those of DNOS where applicable.

Many commands prompt for TERMINAL ACCESS NAME. The proper response to this prompt is of the form STXX or a synonym for STXX. Strictly speaking, the reference is to a particular communication port rather than to a particular terminal. However common usage dictates use of the phrase TERMINAL ACCESS NAME. Note that multi-drop circuits are not supported. Thus no ambiguity arises.

A synonym will be maintained for TERMINAL ACCESS NAME and displayed as the default after initial definition.

All commands set a value in $$CC and $$MN before terminating. All tasks use the DNOS message handling convention.

The following commands are supported:

a. CALL - CALL TERMINAL

b. ANS - ANSWER INCOMING CALL

c. DISC - TERMINAL DISCONNECTION

d. MHPC - MODIFY HARDCOPY TERMINAL PORT CHARACTERISTICS

e. LHPC - LIST HARDCOPY TERMINAL PORT CHARACTERISTICS

Details about these commands can be found in the <u>DNOS System Command Interpreter (SCI) Reference Manual</u>.

All DNOS functions described in the remainder of this section execute as foreground tasks or within background batch streams with the exception of the TPDISC task. The TPDISC task cannot be successfully run from a batch stream that was initiated at a remote terminal.

11.3 TELEPRINTER DEVICE TASKS

Four tasks support the teleprinter device utilities. The task named TPCALANS processes the CALL and ANS commands. The TPDISC task handles the DISC command, TPMHPC handles the MHPC command, and TPLHPC processes the LHPC command.

11.3.1 TPCALANS.

The CALL command is used to establish a connection with another terminal when the call is to be initiated by the system.

The TPCALANS task accepts tuning parameters for the CALL command in the following order. Parameters apply to auto dialing unless noted otherwise. These are specified in the PARM list of the CALL PROC.

1. Max delays in system time intervals

   a. between digits

   b. for primary dialtone

   c. for secondary dialtone

   d. for answerback (auto dial or manual dial)

   e. between DSS true and DSR true

   f. between DPR true and PND false

2. Max delays in seconds

   a. waiting for Data Link Occupied to go low at the start of dialing

   b. waiting for connect (auto dial)

   c. waiting for connect (manual dial - zero means infinite)

3. Time delay in system time intervals between setting CRQ and DTR

4. Time delays in seconds

   a. between a failed attempt and a retry

   b. between Data Link Occupied going low and assertion of Call Request

   c. between receipt of primary dialtone and dialing the first digit

   d. between receipt of secondary dialtone and dialing the next digit

5. Max number of tries

   a. to establish a valid connection

   b. to read an answerback (auto dial or manual dial)

6. Flags (zero = no, not zero = yes)

   a. use "end of number" after last digit

   b. use pulse dialing if using TI internal ACU

   c. use even parity testing on answerback (zero = no parity test)

   d. assert RTS when dialing on half duplex circuits (auto dial or manual dial)

e. save answerback in ABM even if verification not requested

When a number has been dialed, the program delays until either the time interval specified in 2b elapses or the hardware Abandon Call and Retry timer expires. Upon occurrence of either, the call is terminated.

Many countries have laws rigidly controlling computer dialing. The many tuning parameters allow this function to be configured for legal conformity in those countries.

The CODE argument of the .BID statement in the PROC is required and must equal zero when bidding TPCALANS to perform a CALL operation.

The ANS command is used to monitor for completion of a connection with another terminal when the call is to be initiated by the terminal. Whether or not ANS is used, the system is continuously monitoring for a ring signal on those ports which are available. This command alerts the user that a specific incoming call has been received.

The following items are tuning parameters for the ANS command listed in the order indicated.

1. Max delay in system time intervals for answerback
2. Max delay in seconds waiting for connect (Zero will be interpreted as infinite delay.)
3. Number of tries to read an answerback
4. Use even parity testing on answerback (zero = no parity test, not zero = yes)
5. Save answerback in ABM even if verification not requested

When bidding this task to perform ANS, the CODE parameter on the .BID must be set equal to 1.

11.3.2 TPDISC.

The DISC command is used to terminate a connection. Disconnect is allowed provided no task is currently active with respect to the terminal or, if the terminal is ME, only SCI and the TPDISC task are active. If the terminal is ME, then the TPDISC task may not be running in background (i.e. in a batch stream.) If the terminal is ME, this task forces QUIT processing. If other users have LUNOs assigned to the terminal, disconnect is also disallowed.

11.3.3 TPMHPC.

The TPMHPC task allows a user access to the port characteristics modification calls supported by the DSR. Note that this task does not support all possible parameter modifications. Those not supported are generally associated with use of the DSR to drive nonstandard terminals or some other mode of usage which is not encouraged. The user who has need to perform modifications not supported by this task must develop a program to issue the appropriate SVC calls.

11.3.4 TPLHPC.

The TPLHPC task is used to obtain a table of the port characteristics for all hardcopy terminal ports. The table is written to the specified output access name. If no output access name is supplied, the table is written to the TLF.

11.4 HARDWARE ENVIRONMENT

The teleprinter devices use DSRTPD which allows operation in the following environments.

I/F CARD	MODEM OR HARDWIRE	TERMINAL	OPTIONAL ACU
TTY	HARDWIRE	743/5	NO
TTY	HARDWIRE	820KSR	NO
TTY	HARDWIRE	840KSR	NO
COMM/TTY	103J/212A	743/5	EXTERNAL 801C
COMM/TTY	103J/212A	763/5	EXTERNAL 801C
COMM/TTY	103J/212A	781/3/5/7	EXTERNAL 801C
COMM/TTY	103J/212A	820KSR/RO	EXTERNAL 801C
COMM/TTY	103J/212A	840KSR/RO	EXTERNAL 801C
COMM	202S	763/5	EXTERNAL 801C
COMM	202S	781/3	EXTERNAL 801C
COMM	INTERNAL 202	763/5	INTERNAL
COMM	INTERNAL 202	781/3	INTERNAL
9902 PORTS	103J/212A	74x/76x/78x/70x	EXTERNAL 801C

The following terminals will operate as the target terminal for the teleprinter device utilities.

1. 703

2. 707

3. 743

4. 745
5. 763
6. 765
7. 781
8. 783
9. 785
10. 787
11. 820KSR
12. 820RO
13. 840RO

Any terminal supported by DNOS as an SCI terminal may be used to initiate the TPD utilities to interact with a valid target terminal.

SECTION 12

DEBUGGING TOOLS

12.1 OVERVIEW

DNOS provides the following utilities for debugging user programs:

* Debugger - an interactive debugging facility that consists of three types of commands. One set can be used on any task. Another set can be used only on tasks that are executing in a special mode called debug mode. The third set of commands are for programs written in Pascal.

* LLR - a utility that allows the user to list the contents of a record or records in a file. The data is displayed in both hexadecimal and ASCII formats.

* MRFSRF - a utility that allows the user to show or modify data at an absolute word address within a file.

* MPISPI - a utility that allows the user to show or modify a program (defined to be a task segment, a procedure segment, or an overlay) in a specified program file.

12.2 DEBUGGER

The Debugger is a task written in assembly language. It is invoked by any of a set of Debugger SCI commands (See Table 12-1 through Table 12-3). The general commands shown in Table 12-1 can be used on any task, whether or not it is in debug mode. The commands shown in Table 12-2, can only be used on controlled tasks. A controlled task is one that has been put in the debug mode by execution of the Execute in Debug Mode (XD) command.

Table 12-3 is a summary of the Pascal debugging commands. These commands are for debugging object from the Pascal compiler. A Pascal task is put in debug mode by execution of the XD command. The Pascal debugging commands are not covered in detail in this document.

The Debugger commands do not operate in a "closed" environment. Any SCI command is available while using the debug commands since the Debugger is an RBID task.

In this discussion of the Debugger, when the word task is used, without "controlled" or "executing in debug mode", the intention is to refer to a task that is not a controlled task.

With regard to a task that is not executing in the debug mode, the user can halt and resume a task, modify memory, set and reset breakpoints, display words and bytes of memory, evaluate expressions, and modify workspace and internal registers. A breakpoint stops the execution of a task when the program counter (PC) has a specified value.

When debugging a task that is not in the debug mode, the user must supply addresses as absolute addresses or expressions using absolute addresses.

The debug mode allows the user to simulate tasks, and to set and reset simulated breakpoints. These capabilities are in addition to the capabilities available with a task that is not in the debug mode. Simulated breakpoints are based on the occurrence of an event of one of the following types:

* Memory (addresses within a specified range) is altered.

* The communications register unit (CRU) address has a specified value.

* The PC has a specified value.

* Memory (addresses within a specified range) is referenced (either a read or a write).

* The status register (ST) has a specified value.

* A level 15 XOP (SVC) is executed.

When a task is in the debug mode, and a symbol table is available, the Debugger evaluates expressions that contain labels as well as absolute addresses. These expressions determine the addresses used in processing Debugging commands.

A major distinction between executing in debug mode and not in debug mode is that the task is partially simulated in debug mode. The following instructions are simulated:

* Any instructions that alter the flow of control - branches, jumps, returns

* Any instructions which reference or alter memory relocations are partially decoded. This allows

breakpoints on memory reference or alteration.

If an instruction in a controlled task is not simulated, or only partially simulated, the Debugger places a breakpoint at the address immediately after the instruction. The task is allowed to execute one instruction at a time. Halting and restarting the task for each instruction drastically reduces execution time. In Debug mode, instructions execute at approximately 8 per second.

The Debugger tests for breakpoint events following the execution or simulation of each instruction as long as the PC is within the specified debugging range.

The task that is not in debug mode is allowed to run until it executes a PC breakpoint (that is, the Debugger suspends itself and allows the task to execute). In debug mode, the Debugger task remains active and either simulates each instruction as though the controlled task is executing, or allows the task to execute one instruction at a time.

The words TRAP and BREAKPOINT are used in the code comments interchangeably. The word breakpoint is used exclusively in this discussion.

Table 12-1 Debugger General Commands

SCI Command	Command Name	CODE	Module
Data Display Commands			
LB	List Breakpoints	06	D$$LB
LM	List Memory	00	D$$LM
LSM	List System Memory	00	D$$LM
SIR	Show Internal Registers	05	D$$SI
SP	Show Panel	09	D$$SP
SV	Show Value	15	D$$SV
SWR	Show Workspace Registers	0B	D$$SW
Data Modification Commands			
MIR	Modify Internal Registers	0A	D$$MI
MM	Modify Memory	04	D$$MM
MSM	Modify System Memory	04	D$$MM
MWR	Modify Workspace Registers	14	D$$MR
Breakpoint Commands			
AB	Assign Breakpoint(s)	01	D$$AB
DB	Delete Breakpoint(s)	02	D$$DB
DB(ALL)	Delete All Breakpoints	1F	D$$DAB
DPB	Delete/Proceed from Breakpoint(s)	03	D$$DP
PB	Proceed from Breakpoint	13	D$$PB
Task Control Commands			
AT	Activate Task	--	-----
HT	Halt Task	07	D$$HT
QD	Quit Debugger	10	D$$QD
RT	Resume Task	08	D$$RT
XD	Execute in Debug Mode	0D	D$$DEB
XHT	Execute and Halt Task	--	------
Search Commands			
FB	Find Byte	16	D$$FB
FW	Find Word	17	D$$FW

Table 12-2 Debugger Commands for Controlled Tasks

SCI Command	Command Name	CODE	Module
ASB	Assign Simulated Breakpoint(s)	0C	D$$ASB
DSB	Delete Simulated Breakpoint(s)	0E	D$$DSB
LSB	List Simulated Breakpoints	0F	D$$LSB
RST	Resume Simulated Task	11	D$$RS
ST	Simulate Task	12	D$$S
XD	Execute Simulated Debug		D$$DEB
QD	Quit Simulated Debug		D$$QD

Table 12-3 Pascal Debugger Command Summary

SCI Command	Command Name	CODE	Module
ABP	Assign Breakpoint(s) - Pascal	1A	D$$APB
DBP	Delete Breakpoint(s) - Pascal	1B	D$$DPB
DPBP	Delete/Proceed from Breakpoint(s) - Pascal	1C	D$$DPP
LBP	List Breakpoints - Pascal	1E	D$$LRB
LPS	List Pascal Stack	19	D$$LPS
PBP	Proceed from Breakpoint - Pascal	1D	D$$PPB
SPS	Show Pascal Stack	18	D$$SPS

12.2.1 Operating System Considerations.

The Debugger implements a breakpoint by replacing the instruction at the specified address with >2FCF, the object for the following assembly language instruction:

```
XOP      15,15
```

The Debugger is designed with the expectation that the XOP processor at level 15 performs the following services:

* Recognition that the call is from a task that has had the >2FCF inserted by the Debugger. This is done in RPROOT, the level 15 XOP processor for DNOS. RPROOT determines that the call block was in register 15 in the workspace of the requesting task, and that the instruction is a level 15 XOP. (Register 15 need not

contain an Unconditional Suspend SVC call block. Any task that executes the >2FCF instruction is suspended without examination of the call block.)

* Adjust the PC of the requesting task backward to repeat the instruction at this address. The command processors that delete breakpoints and proceed from breakpoints are designed on the assumption that the PC is adjusted before they are called.

* A flag in the TSB is set to indicate the task is suspended because of execution of a breakpoint. This flag is cleared by the Resume Task SVC processor.

Following this processing, the operating system marks the task suspended and reactivates the parent task. In DNOS, SCI990 is always the parent task of the task being debugged. SCI990 RBIDs the Debugger when a Debugger command is processed.

The Debugger issues an SVC to extend the time slice prior to altering the breakpoint table. This is not required in DNOS, but is done to remain compatible with DX10. In DX10, the breakpoint table is in the operating system address space, and the Debugger must complete the alteration of the table without interruption in order to ensure the integrity of the breakpoint table.

12.2.2 Structure of the Task.

The Debugger consists of a task segment and a procedure segment. It calls routines in the S$SYSTEM shared procedure segment. The Debugger is a replicatable, hardware- and software-privileged task. It must be privileged so that it can simulate privileged instructions. The Debugger issues the following privileged SVCs in its own behalf:

* Read/Write Task

* Read/Write TSB

The Debugger task segment contains the transfer vector, the driver routine D$OV1, frequently used routines and global data.

The data and workspace modules are as follows:

* DW$OV1 - workspaces

* DD$COM - Common data area for the Debugger. This module contains declarations for buffers, pointers, and tables used throughout the processing of commands.

* DBGTSK - IRBs, related data structures, and miscellaneous declarations for the Debugger

* PARSED - Data segment

* Four workspaces for S$SYSTEM routines

* D$SDAT - Simulator local data area

* DL$OV1 - local data area

* DL$RWT - call blocks (IRBs) for SVCs that read and write data in the address space of the task being debugged.

* DQ$GEN - Equates, using set prefixes in six-character names. The characters after the prefix are optional, and are as meaningful as two or three characters can be. The forms of the equate names are as follows:

  - DE$abc - error codes

  - TSBabc - TSB displacements

  - TS$abc - Task states

  - TSFabc - TSB flag bits

  - EC$abc - Event keys

  - DQ$abc - Breakpoint table locations and displacements, system pointers, IRB opcodes, and miscellaneous data

  - TRAPab - Simulated breakpoint table displacements

  where:

  a, b, and c are nonblank alphanumeric characters.

* D$MSG - Text of all messages written by the Debugger to the user screen

The following program modules are in the Debugger task segment:

* D$OV1 - Routine D$OV1, the driver for the Debugger

* D$RTS - Routine D$RTS, read TSB

* D$RTW - Routine D$RTW, read a word in the task address space

12.2.3 Flow of Control.

Once the Debugger task has been activated (by a Debugger command), it remains in the user's job until both of the following conditions are met:

* There is no controlled task associated with the Debugger task. That is, the internal variable DD$CTI has a value of zero. This variable is set to the run ID of the controlled task by the XD command processor, and is given a value of zero by the QD command processor. The task being debugged can terminate and no longer be active in the user's job, but the QD command is still required to clear DD$CTI.

* All breakpoints assigned by the Debugger in any task in the user's job are deleted. This condition is monitored in the driver routine through a local variable. The value of BRKPNT is the number of breakpoints that are set, but not deleted.

If both conditions are satisfied after the completion of command processing, the Debugger exits through S$TERM. Otherwise, the Debugger exits through S$WAIT and is suspended.

12.2.3.1 Invoking the Debugger.

The Debugger is bid (or reactivated) by SCI while processing a Debugger command procedure. The .RBID statement contains the CODE and a PARMS list that varies, according to the command.

12.2.3.2 Initialization.

The following initialization steps are repeated each time the Debugger is RBID:

1. Open the terminal LUNO, with event characters enabled.

2. Initialize the VDT field size.

3. Initialize the local flag for the use of 8-bit ASCII, based on the country code.

4. Get access to the TCA and the status of the terminal.

Routine D$$DEB establishes the environment for a controlled task. This initialization consists of the following steps:

1. Set the controlled bit in the TSB.

2. Initialize DD$CTI to the run ID of the task that is to run in debug mode.

3. Initialize instruction count, WP, PC, and ST. It is the limit on the number of instructions to be executed in debug mode.

4. Initialize the 990/12 flag to one of the following values:

   0: 990/10 object, 990/12 instructions not supported

   1: 990/12 object

5. Call D$BST to initialize task symbols. This routine extends the Debugger task space and constructs a local symbol table for use in evaluating symbolic expressions.

12.2.3.3 Major Loop/Routines.

D$OV1, the driver, is a table driven algorithm. D$OV1 derived its name from being an overlay on DX10. The special cases are handled first. If the CODE value passed is >00FF, a branch to the end-action routine in D$OV1 is taken. This is not the same processing as is done with the QD command. If the session is in batch mode, an error is returned (through S$STOP) for any request other than List Memory or List System Memory.

The CODE value is an index into the command processor address table. A branch is taken to the appropriate address for command processing.

12.2.3.4 Error Processing.

The Debugger reports some errors to the user through SCI. The termination synonyms are set to reflect the condition. The error codes returned are defined in module DQ$GEN.

Module D$MSG contains error messages and informative messages. These messages are output directly to the user, avoiding the SCI interface. The debugger uses these messages when it does not want to suspend itself by returning to SCI. An example is the messages supplied during task simulation.

12.2.3.5 Termination.

The Debugger terminates by calling S$STOP. When the debug mode is active, the Debugger terminates only when processing the QD (Quit Debugger) command. If there is no pending debug activity the Debugger terminates after processing each request. Whether or not there is pending debug activity is determined by examining the values of the local variables DD$CTI and BRKPNT.

12.2.4 Data Structures.

Module DL$VEC contains three vectors associated with user interface. The name of the data structure is KWT. A maximum of 22 entries is supported. The entries are called keywords in the code, but their function is that of SCI field prompt. The following information is maintained for each of the entries:

* Text of the prompt - five bytes, called the keyword vector

* Default value - the current value

* Actual value - the value entered by the user in response to the prompt.

The data structure OPTABL in module D$SINS contains the opcodes, text strings, and a flag word for every 990/10 and 990/12 assembly language instruction. The flag word contains the following information:

* Hardware- or software-privileged instruction

* 990/12 instruction

* Illegal op code

* Instruction group - The instruction group is an index into the address table (in D$SIM) for simulation or execution of the instruction. Instructions are grouped according to execution similarity, allowing one piece of code to simulate many instructions.

The TCA data structure (synonym and logical name segment in DNOS) is transparent to the Debugger since all access of the PARMS list is done by calling S$PARM. The other system data structures utilized by the Debugger are the task status block (TSB), and the system pointer table (NFPTR). The TCA, TSB, and NFPTR are documented in the Data Structures Pictures section of the DNOS System Design Document.

The variable DD$CTI is the task run ID of the controlled task. The special value of zero for DD$CTI is used throughout the Debugger code to mean that there is no controlled task.

The constant DQ$SSU is the user privilege level required to perform Debugger functions in the system address space. In the current release, the minimum privilege level required is two.

Task breakpoint information is stored in the data structure D$BRKP, in module DBGTSK. The table consists of 32 six-byte entries, each of which contains the following data:

* Run ID of the task in which the breakpoint occurs
* Address of the breakpoint
* Original contents of the breakpoint location

Simulated breakpoint information is stored in the data structure EVENTS, in module DD$COM. The simulated breakpoint table has space for ten six-word entries, each of which contains the following data about the breakpoint:

* Running count of the number of times the breakpoint event has occurred.
* Starting memory address for display when the breakpoint is executed
* The first address of the breakpoint range
* The last address of the breakpoint range
* The number of times the event is to occur before the breakpoint is executed
* A code that indicates the event on which the breakpoint is based. The codes are as follows:
  - P: PC value
  - S: ST value
  - C: CRU value
  - R: Memory reference
  - A: Memory access
  - X: XOP 15

TYPTRP is a table of codes for the events on which breakpoints can be based.

The Debugger maintains a simulated context (WP, PC, and ST). The simulated context is updated after each simulated instruction. Before simulation, the simulated context is compared to the WP, PC, and ST of the task's TSB. If they are different the simulation task is set to the TSB's context. This guarantees a correct simulation starting state.

The Debugger maintains a local data structure that contains values for symbols. The address space of the Debugger is expanded by a Get Memory SVC to store the symbol table. SYMTBL is the local pointer to the table.

The logic of the symbol compression algorithm, similar to the assembler compression algorithm, is shown in the following metacode:

```
D$CMP symbol compression algorithm
   R2=0;
   R5=0;
   I=1;
   DO UNTIL (I=8) OR (no more characters in name);
     Load next character into R2;
     IF character is non-blank
       THEN R2 = exclusive OR of character and R5;
            Shift circular R2 five bits;
     I = I + 1;
   END DO;
```

12.2.5 Files.

The files accessed by the Debugger are the terminal local file (TLF) and the linked object file that may be optionally specified with the Execute Debug (XD) command. The linked object file is read to build the internal symbol table that is used to evaluate symbolic expressions.

12.2.6 Synonyms.

The Debugger uses the synonym $$DA to indicate whether or not the debug mode is active. $$DA is set to a value of YES when a task is executing in the debug mode.

In DX10, the SCI variable S$$MNU is altered to suppress the display of the default menu. The code to alter S$$MNU is also in the DNOS Debugger, but the variable is local to the Debugger task and has no effect on SCI. In DNOS, surpressing the display of the main menu is accomplished via the .MENU command. The .MENU is placed at the end of the debugger command procedures.

12.2.7 Coding Conventions.

The Debugger code does not follow DNOS coding conventions. The major exceptions are as follows:

* Labels do not start with a fixed prefix.

* Error equates are not of the format ERRaa.

* Prologues are incomplete.

12.2.8 Subroutine Linkage.

The command processors and the subroutines they call have a standard BLWP/RTWP interface.

The RSTACK area in module DBGTSK is not used by the Debugger code.

12.2.9 Detailed Design.

This discussion of the detailed design of the Debugger includes a paragraph on the driver routine, a paragraph on each of the command processors, and a paragraph detailing some low-level routines called by many of the processors. The command processors are discussed in the same order as in Table 12-1 and Table 12-2.

The driver routine for the Debugger is D$OV1. The routines invoked by D$OV1 all have names of the following form:

D$$abc

where:

abc is the SCI command name, except for the XD command. Routine D$$DEB processes the XD command.

In general, these routines are small. They typically process the PARMS list and the CODE value on the .RBID statement, and then invoke lower level routines that do the actual processing. These lower level routines have names that begin with the letters D$.

Modules containing the source for the Debugger are in the directory DSC.DEBUGGER.SOURCE, where DSC is the name of the directory in which the DNOS source, object, and assembly listings reside.

12.2.9.1 D$OV1.

The first function of D$OV1 is initialization, which consists of the following steps:

1. Issue a Self-ID SVC to get the run time task ID (of the Debugger) and the station ID from which it is RBID.

2. Open the terminal LUNO, with event characters.

3. Issue a Read Device Characteristics operation of the

I/O SVC to determine the size of the display, that is, the number of lines and the number of columns in each line.

4. Based on the NFDATA variable for country code, initialize the following language flags:

   - Set the eight-bit ASCII flag if the country code is Japanese.

   - Set the low and high Arabic values if the country code is Arabic.

   - Set the eight-bit display flag for Japanese or Arabic country code.

5. Call S$GTCA to gain access to the communication area (TCA), that is, the synonym segment and the logical name segment.

6. Call S$STAT to get the terminal status and the value of the CODE from the .RBID statement.

The CODE value is examined to ensure that it is a valid request. In batch mode, the only CODE value allowed is a value of zero. D$OV1 logic includes the assumption that a CODE value of zero is for either the List Memory (LM) or the List System Memory (LSM) command.

D$OV1 uses the CODE value to index into the command processor address table. The debugger branches to the entry point of the processor found in the table.

After completion of command processing, D$OV1 calls S$SETS to assign the appropriate value to the synonym $$DA (Y or N). D$OV1 calls S$WAIT if the Debugger is to be suspended and calls S$STOP if the Debugger is to terminate.

12.2.9.2 List Breakpoints.

SCI Command: LB

MODULE: D$$LB

CODE: >06

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the list breakpoint command processor is D$$LB in module D$$LB. The function of D$$LB is to produce a listing of all breakpoints that are set through either the AB or DPB commands. The logic of D$$LB is shown in the following metacode:

```
D$$LB:
   Call D$PR1 to process the run ID;
   Initialize run ID and display line number;
   IF terminal mode is VDT
     THEN Call D$CSR to clear the screen;
          Call D$$WL to write the header line;
   LOOP:
      DO UNTIL (no more breakpoints);
         Increment line number;
         Call D$BBL to build display line for breakpoint;
         Call D$WL to write line to display;
   END LOOP;
   Modify S$$MENU to preserve breakpoint display;
```

D$BBL reads the breakpoint table and constructs the display line. The length of the buffer passed to D$BBL determines the number of breakpoints that can be listed on one line. When there are as yet unlisted breakpoints and the buffer is full, a + is inserted at the end of the buffer (one character at the end of the buffer is reserved for this purpose). D$BBL calls S$IASC to convert binary addresses to ASCII characters. The only information listed about the breakpoint is the address.

12.2.9.3 List Memory, List System Memory.

SCI Commands: LM, LSM

MODULE: D$$LM

CODE: >00

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged (LM)	RUN ID
1	Overlay ID (LSM)	OVERLAY NAME OR ID *
2	Starting address of memory area to be listed	STARTING ADDRESS
3	Number of bytes to list	NUMBER OF BYTES
4	Listing access name	LISTING ACCESS NAME
5	System memory indicator Null: LM Non-null: LSM	None

* The overlay name is translated by the command procedure to the installed overlay ID in the kernel program file.

The entry point for the list memory and list system memory command processor is D$$LM in module D$$LM.

D$$LM processes both the List Memory and the List System Memory commands. The function of D$$LM is to produce a listing of memory starting at a specified address and continuing a specified number of bytes. The logic of D$$LM is shown in the following metacode:

```
D$$LM:
   IF fifth PARMS element is non-null
     THEN Call D$OID to get the overlay ID;
          Set system memory flag;
     ELSE Call D$PR2 to process the run ID;
   Call D$PSP to process the beginning address;
   Call D$PSP to process the ending address;
   IF PARM four is null
     THEN Clear pointer to listing file access name;
   IF request is to list system memory
     THEN Call D$LM;
     ELSE Call D$HT to halt the task;
          Call D$LM to list memory;
          Call D$RST to restore the original state
               of the task;
```

D$OID enforces a minimum privilege level for listing system memory. It calls S$STAT to get the status of the user, and if the privilege level is greater than or equal to variable DQ$SSU, processing continues. This testing against DQ$SSU is described in the code comments as a test for system user, but it is a privilege level check only. The value of DQ$SSU in the current release of DNOS is 2.

D$OID gets the first element of the PARMS list (the overlay ID) and converts it to a binary number. The overlay ID is used in the call block for the Read/Write Task SVC.

12.2.9.4 Show Internal Registers.

SCI Command: SIR

MODULE: D$$SI

CODE: >05

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the show internal registers command processor is D$$SI in module D$$SI. If the terminal is in TTY mode, the current workspace registers are displayed on a single line. If the terminal is in VDT mode, the entire debug panel is displayed. The logic of D$$SI is shown in the following metacode:

```
D$$SI:
   Call D$PR1 to process the run ID;
   Call D$SPT to suspend the task;
   Call D$PIR to process the internal registers display;
   Call D$RST to restore the state of the task;
```

12.2.9.5 Show Panel.

SCI Command: SP

MODULE: D$$SP

CODE: >09

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID
2	Address of the first word of memory to be displayed	MEMORY ADDRESS

The entry point for the show panel command processor is D$$SP in module D$$SP. The function of D$$SP is to build the debug panel display and write it to the interactive terminal. The logic of D$$SP is shown in the following metacode:

```
D$$SP:
   Call D$PR1 to process the run ID;
   Call D$PSP to process the address parameter;
   Round address down to even value;
   IF address parameter is null
     THEN Use >FFFF, meaning current PC;
   IF run ID is the controlled task;
     THEN Update its memory address;
   Call D$HT to halt task;
   Call D$POP to put out the panel;
   Call D$RST to restore the state of the task;
```

12.2.9.6 Show Value.

SCI Command: SV

MODULE: D$$SV

CODE: >15

PARMS:

PARM Number	Definition	Field Prompt Name
1	The expression to be evaluated	EXPRESSION

The entry point for the show value command processor is D$$SV in module D$$SV. The function of D$$SV is to evaluate the expression and display the value in the following forms: hexadecimal, decimal, and ASCII. The logic of D$$SV is shown in the following metacode:

```
D$$SV:
   Call D$CSR to clear the screen;
   Call S$PARM to get the expression;
   Call D$ESE to evaluate the expression;
   Build the output line;
   Call S$IASC to convert hexadecimal value to ASCII;
   Replace the byte count with >;
   Call S$IASC to convert decimal value to ASCII;
   Insert labels and dots in output buffer;
   Replace dots with characters when printable;
   Call D$WL to write the line to the display;
```

D$$SV contains code to alter the variable S$$MNU. In DNOS, the command procedure must preserve the display.

12.2.9.7 Show Workspace Registers.

SCI Command: SWR

MODULE: D$$SW

CODE: >0B

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the show workspace registers command processor is D$$SW in module D$$SW. The function of D$$SW is to format and write to the terminal the sixteen workspace registers from the current context of the specified task. The logic of D$$SW is shown in the following metacode:

```
D$$SW:
   Call D$PR1 to process the run ID;
   Call D$HT to halt the task
   IF terminal mode is VDT
     THEN Call D$POP to put out the panel;
     ELSE Call D$RTS to get WP from TSB;
          Call D$DM to display workspace memory;
   Call D$RST to restore state of task;
```

12.2.9.8 Modify Internal Registers.

SCI Command: MIR

MODULE: D$$MI

CODE: >0A

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the modify internal registers command processor is D$$MI in module D$$MI. The function of D$$MI is to display current values of internal registers (WP, PC, and ST) to the user and to modify the contents of the registers according to input from the terminal. D$$MI interfaces with the user terminal in a manner similar to SCI. Routine S$FMT is called to display the current values. What are called field prompts in SCI are called keywords in the code comments of D$$MI. The text displayed to the user is in the data structure KWT in module D$MSG. S$GKEY is called to return user input to D$$MI. The logic of D$$MI is shown in the following metacode:

```
D$$MI:
  Call D$PR1 to process the Run ID;
  Call D$HT to halt the task being debugged;
  DO for all keywords;
    Convert keyword value in current TSB to ASCII and store;
  END DO;
  Call S$FMT to clear the screen and declare screen format;
  Initialize keyword counter;
  LOOP:  DO UNTIL (keyword counter = 3) or (CMD key entered);
    Call S$GKEY to get user value for field prompt;
    IF error from S$GKEY
      THEN Call D$PKE to process error;
      ELSE IF event key
           THEN IF CMD key
                  THEN Exit LOOP;
                IF EOL, TAB, or SKIP
                  THEN Process as a RETURN (at LABEL1);
                  ELSE Set error message indicator for S$GKEY;
                       GO TO LOOP;
  LABEL1:  ELSE IF value entered is non-null
                  THEN Call D$PSE to process expression;
                       IF Error
                         THEN Set error message code for S$GKEY;
                              GO TO LOOP;
                         ELSE IF new value is not old value;
                                THEN IF status register
                                       THEN Enforce privilege
                                            level rules;
                                       ELSE IF PC register
                                              THEN Reset break-
                                                   point bit;
                                     Put new value into TSB;
                              ENDIF;
                ENDIF;
  Increment keyword counter;
  END LOOP;
  Call D$PRP to process the panel;
  Call D$RST to restore state of the task;
```

D$PKE allows the caller to continue execution only if the error code being examined is between >9000 and >9FFF, inclusive. If the error code is in that range, a pointer to the error message INVALID EXPRESSION (or the local language equivalent) is loaded in the appropriate register for display by S$GKEY the next time it is called. Otherwise, the error is considered irrecoverable and D$PKE returns to the caller through an error exit.

D$PSE requires as inputs the Run ID and the address of a string that is a symbolic expression. It returns the value of the expression or the address of an error message in the caller's workspace. Register assignments for the interface are documented in the code. D$PSE calls D$ESE to evaluate the expression.

12.2.9.9 Modify Memory, Modify System Memory.

SCI Commands: MM, MSM

MODULE: D$$MM

CODE: >04

PARMS:

PARM Number	Definition	Field Prompt Name
1-MM	Run ID of task being debugged	RUN ID
1-MSM	Overlay ID	OVERLAY NAME OR ID *
2	First memory address to be modified	ADDRESS
3	System memory indicator Null implies not system Non-null for system	None

* The overlay name is translated by the command procedure to the installed overlay ID in the kernel program file.

The entry point for the modify memory and modify system memory command processor is D$$MM in module D$$MM. The function of D$$MM is to display the contents of specified locations in the address space of a specified task, and to interface with the user to alter the contents of those locations. The logic of D$$MM is shown in the following metacode:

```
D$$MM:
   Call S$PARM to get third PARM;
   IF PARM is null
      THEN Call D$PR2 to process Run ID;
      ELSE Call D$OID to process Overlay ID;
           Set system memory flag;
   Call D$PSP to get and process the beginning address;
   Round address down to an even number;
   IF not system memory
     THEN Call D$HT to halt the task;
          IF Run ID is not zero
            THEN Call D$BIR to build register display;
   DO UNTIL (CMD key is input);
     Initialize local memory address;
     Clear line count;
     DO UNTIL (line count = max) OR (error in fetching memory);
       Initialize pointers, buffers, and flags;
       Call D$GOC to get 8 words of memory;
       IF error
         THEN IF not end of memory
                THEN Take error exit;
       Clear forced fetch flag;
       Add number of words read to line count;
       IF zero words read
         THEN Take error exit;
         ELSE IF line count > number of words read
                THEN Start another line of output;
                     Call D$GOC for second half;
                ELSE Use second half of display screen;
              Format output lines;
              Add number of words read to line count;
     END DO;
   Call S$FMT to write current values to screen;
   Initialize keyword count;
   LOOP:  DO UNTIL (keyword count = line count);
   Call S$GKEY to read terminal;
   IF error
     THEN Call D$PKE to process it;
          Set error message pointer for S$GKEY;
          GO TO LOOP;
   IF event character
     THEN IF CMD key
            THEN Exit LOOP;
          IF EOL, TAB, or SKIP
            THEN Process as a RETURN (at LABEL1);
            ELSE Set error message indicator for S$GKEY;
                 GO TO LOOP;
```

```
LABEL1:
  ELSE IF value entered is non-null
         THEN Call D$PSE to process symbolic expression;
              IF Error
                THEN Set error message number for S$GKEY;
                     GO TO LOOP;
                ELSE Compare new value with old value;
                     IF they differ
                       THEN Call D$WTW to update task
                                  address space;
                     Update value vector for display;
       ENDIF;
Increment keyword counter;
Increment base memory address;
END LOOP;
IF modifying system memory
  THEN Exit;
Call D$PRP to process the panel
IF modifying system memory
  THEN Exit;
Call D$RST to restore the state of the task;
```

A zero value for the forced fetch flag (FFFLAG) is a signal to D$GOC that all or part of the octet being requested may still be in memory in the call block from a Read/Write Task SVC previously issued. The two routines are interdependent in that D$GOC takes the forced fetch flag from the caller and assumes it is valid. D$$MM is written with the assumption that D$GOC builds the SVC call block such that it reads 16 words.

The logic of D$$MM requires that the buffers OCTET1 and OCTET2 be contiguous, and that OCTET2 follow OCTET1.

12.2.9.10 Modify Workspace Registers.

SCI Command: MWR

MODULE: D$$MR

CODE: >14

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID
2	First workspace register to be displayed (and potentially modified)	REGISTER NUMBER

The entry point for the modify workspace registers command processor is D$$MR in module D$$MR. The function of D$$MR is to display the contents of the current workspace registers, and to interface with the user to alter the contents of those registers. The logic of D$$MR is shown in the following metacode:

```
D$$MR:
   Initialization;
   Call D$PR2 to process run ID;
   Call S$PARM to get second PARM;
   Call D$ESE to evaluate the PARM;
   Convert register number to byte displacement;
   Call D$HT to halt task being debugged;
   IF run ID is not 0
     THEN Call D$BIR to build display;
   Call D$RTS to get value of workspace pointer register from TSB;
   Initialize workspace buffer;
   DO UNTIL (CMD key);
     DO UNTIL (past register 15) OR (error fetching memory);
       Call D$RTW to read a word in task memory space;
       IF error
         THEN IF line count is zero
                THEN Abort;
       Increment line count;
       Store value read;
       Call S$IASC to convert value to ASCII;
       Insert > into field;
       Call S$SCPY to store string in keyword buffer;
       Adjust pointers;
     END DO;
     Call S$FMT to format the screen;
     Initialize keyword pointer;
```

```
LOOP:
  DO UNTIL (CMD key) OR (keyword returned = line count);
PROMPT:
  Call S$GKEY to read terminal;
  IF error
    THEN Call D$PKE to process error;
         Clear error code;
         GO TO PROMPT;
  IF event character
    THEN IF CMD key
           THEN Exit LOOP;
         IF EOL or TAB or SKIP
           THEN Process as a RETURN at LABEL1;
           ELSE Load error message address;
                GO TO PROMPT;
LABEL1:
    ELSE Adjust pointers;
         IF parameter is non-null
           THEN Call D$PSE to process expression;
                IF error
                  THEN Load error message address;
                       GO TO PROMPT;
                Increment line count and memory address;
                Call D$WTW to update task memory;
                Update screen format buffer with new value;
END LOOP;
IF CMD key input
  THEN Call D$PRP to process panel;
Call D$RST to restore the state of the task;
END DO;
```

12.2.9.11 Assign Breakpoint.

SCI Command: AB

MODULE: D$$AB

CODE: >01

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task to be debugged	RUN ID
2-n	Addresses at which breakpoints are to be assigned.	ADDRESS(ES)

The entry point to the assign breakpoint command processor is DP$$AB in module D$$AB. The function of D$$AB is to make entries in local data structures and to alter the task address space to implement a breakpoint at each specified address. The logic of the processor is shown in the following metacode:

```
D$$AB:
   Call D$PR1 to process the run ID;
   Call D$HT to halt the task;
   Call D$ABS to assign the breakpoint(s);
   D$ABS:
         DO UNTIL (End of parameter list, or irrecoverable error);
            Get the next element of the PARMS list;
            IF error from PARMS list
              THEN IF null parameter error
                     THEN Return with no error or duplicate
                          breakpoint error;
                     ELSE Abort attempt to assign breakpoints;
                          Return;
              ELSE Call D$AB to assign the breakpoint;
                  IF error in assigning breakpoint
                    THEN IF error = duplicate breakpoint at address
                           THEN Save error and keep going;
         END DO;
         Call D$PRP to display the breakpoints assigned;
         Call D$RST to restore the state of the task;
            END D$ABS;
```

The logic of routines D$AB and D$MEB is shown in the following metacode:

```
D$AB:
   Extend the time slice;
   Call D$MEB to make entry in breakpoint table;
   D$MEB:
     Search the entire breakpoint table for duplicate entry;
     IF Duplicate entry found
       THEN Set error code for duplicate entry;
            Return;
     Search breakpoint table for available entry;
     IF Available entry found
       THEN Make the entry;
            Increment breakpoint count;
       ELSE Set error code for breakpoint table overflow;
            Return;
     Return;
END D$MEB;
```

12.2.9.12 Delete Breakpoint.

SCI Command: DB

MODULE: D$$DB

CODE: >02

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of the task	RUN ID
2	List of addresses of break-points to be deleted	ADDRESS(ES)

The entry point to the Delete Breakpoint command processor is D$$DB in module D$$DB. The function of D$$DB is to remove breakpoints at the specified addresses. The logic of D$$DB is shown in the following metacode:

```
D$$DB:
   Call D$PR1 to process run ID;
   Call D$HT to halt the task;
   Initialize PARM counter;
   Call S$PARM to get first PARM;
   IF first PARM begins with AL
      THEN Call D$DAB to delete all breakpoints;
      ELSE IF first PARM is null
             THEN Call D$VTB to verify that the task is
                       at a breakpoint;
                  Call D$RTS to get current PC;
                  Call D$DB to delete breakpoint at current PC;
             ELSE DO UNTIL PARM is null;
                   Evaluate PARM;
                   Call D$DB to delete the breakpoint;
                   Increment PARM counter;
                   Call S$PARM to get next PARM;
                  END DO;
   Call D$PRP to process the display panel;
   Call D$RST to restore the state of the task;
```

This routine deletes breakpoints assigned with either AB or DPB. Deleting a breakpoint consists of the following steps:

* Restoring the original contents in the task address space

* Removing the information from the local breakpoint table data structure

* Decrementing BRKPNT, the breakpoint counter

12.2.9.13 Delete All Breakpoints..

SCI Command: DB

MODULE: D$$DAB

CODE: >1F

PARMS: None

The entry point to the Delete All Breakpoints command processor is D$$DAB in module D$$DAB. The function of D$$DAB is to delete all breakpoints from the breakpoint table, removing them from existing tasks. The logic of D$$DAB is shown in the following metacode:

```
D$$DAB:
   DO FOREVER
        Call D$GBA to get breakpoint table address;
        Search table till find an active entry;
        If no entry found THEN exit D$$DAB;
        Get run ID from entry;
        Call D$HT to halt the task;
        If task is not extant THEN set a TNE flag;
   LOOP: DO FOREVER
            Call D$GBE to get an active breakpoint for the task;
            If none found THEN exit LOOP;
            Call D$DB to delete the breakpoint;
            If error on delete and (error is not task-not-extant
                task-not-extant or the TNE flag is not set)
                THEN exit LOOP;
         END DO;
         If TNE flag is not set THEN Call D$RST to restore
                task state;
         If LOOP exit due to error THEN exit D$$DAB;
      END DO;
```

D$$DB will restore the contents of the breakpoint address if the task exists. In any case, it will delete the entry from the breakpoint table.

12.2.9.14 Delete/Proceed from Breakpoint(s).

SCI Command: DPB

MODULE: D$$DP

CODE: >03

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of the task being debugged	RUN ID
2-n	List of breakpoint addresses to be assigned	DESTINATION ADDRESS(ES)

The entry point to the Delete and Procede from Breakpoint command processor is D$$DP in module D$$DP. The function of D$$DP is to delete the breakpoint at the current PC, to assign the specified breakpoint(s), and to reactivate the controlled task if it is suspended. The logic of D$$DP is shown in the following metacode:

```
D$$DP:
   Call D$PR1 to process the run ID;
   Call D$ABS to assign the specified breakpoint(s);
   Call D$VTB to verify that the task is at a breakpoint;
   Call D$RTS to get the current PC;
   Call D$DB to delete the breakpoint at the current PC;
   Call D$PRC to issue an Activate Suspended Task SVC;
   Call D$DMC to wait for the controlled task to reach
    a breakpoint, or for the user to enter a control key;
```

D$DB calls D$EXT to extend the time slice. The time slice extension is necessary in DX10 because the breakpoint table is in the system address space. To preserve the integrity of the table, the alterations must not be interrupted. Extending the time slice is not required in DNOS because the breakpoint table is in the Debugger task segment. D$DB calls D$WTW to restore the original contents in the address space of the task being debugged. D$WTW cancels the extended time slice.

In D$PRC, the call block of the Activate Suspended Task SVC is structured to clear the following flags in the TSB of the task:

* Breakpoint

* SVC trap
* Suspend
* Stopped

If any of the called routines returns an error, D$$DP returns to the control routine with an error code.

12.2.9.15 Proceed from Breakpoint.

SCI Command: PB

MODULE: D$$PB

CODE: >13

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID
2-n	Addresses of new breakpoint(s)	DESTINATION ADDRESS(ES)

The entry point for the Proceed from Breakpoint command processor is D$$PB in module D$$PB. The function of D$$PB is to create breakpoints at the specified addresses and to restart the specified task, executing the instruction which was at the current breakpoint. The logic of D$$PB is shown in the following metacode:

```
D$$PB:
   Call D$PR1 to process the run ID;
   Load the overlay that contains D$PIR and D$PSP;
   IF error
     THEN Return with overlay SVC error message;
   Call D$ABS to assign breakpoint;
   IF error
     THEN Process error;                              *
   Call D$POB to procede over the breakpoint;
   Load overlay that contains D$DMC;
   IF error
     THEN Return with overlay SVC error message;
   Call D$DMC (debug mode controller);
   IF error
     THEN Process error;                              *
   Return;
```

* Errors returned to D$$PB from D$ABS and D$DMC are

processed according to the logic shown in the following metacode:

```
IF read task error
  THEN IF task is in debug mode
         THEN Call D$RSE to reset local variables/tables that
                describe the controlled task;
              Return with error message indicating that the
                controlled task terminated in debug mode;
         ELSE Return with task terminated error message code;
ENDIF;
```

The function of routine D$POB is to simulate the instruction at the breakpoint and then proceed. The instruction is simulated so that if it is a branch or jump instruction, only the one instruction is executed. The logic of D$POB is shown in the following metacode:

```
D$POB:
   Call D$VTB to verify that task is at breakpoint;
   Save current PC, WP, and Status to set up simulation
         environment;
   Set new PC, WP, and Status;
   Call D$DB to delete the breakpoint at current PC;
   Load overlay that contains D$STT;
   Call D$STT to simulate the instruction;
   Call D$AB to restore the breakpoint;
   Restore PC, WP and Status to restore environment;
   Call D$PRC to proceed;
   IF read task error
     THEN IF task is in debug mode
            THEN Call D$RSE to reset local variables/tables that
                   describe the controlled task;
                 Return with error message indicating that the
                   controlled task terminated in the debug mode;
            ELSE Return with task terminated error message code;
   ENDIF;
```

The function of D$STT is to simulate the controlled task to a breakpoint or for the specified number of instructions. The logic of D$STT is shown in the following metacode:

```
D$STT:
   Initialize local variables to control simulation;
   LOOP:  DO UNTIL (breakpoint event occurs);
          IF iteration count is zero
            THEN Set time-out trap code;
                 Return;
          IF PC is out of range
            THEN Set time-out trap code;
                 Return;
          Ensure even PC value;
          Store current context and instruction in registers;
          Call D$SINS to identify instruction;
          Call D$GAS to process from and to addresses;
          CASE (trap type);
            Memory access:  Set memory access trap code;
            Illegal opcode: Set illegal opcode trap code;
            Privileged opcode:  Set privileged opcode trap code;
            PC value:  Call D$CKT to process set trap code;
            ST value:  Call D$CKT to process set trap code;
          END CASE;
          Call D$SIM to simulate instruction;
          IF >EFF address traps set
            THEN GO TO TRAPAF;     return with error;
            ELSE IF WP = >FFFF
                   THEN Process as an X group instruction;
          Clear extended execute flag;
          Decrement iteration counter;
          Write new context to task TSB;
          Reset allow trap variable;
   END LOOP;
```

D$CKT is a local subroutine (in module D$STT). It searches the simulated breakpoint table to see if the current event matches a breakpoint condition. In order to meet the criteria for a PC or ST breakpoint event, the breakpoint type must match and the PC must be within the breakpoint range of addresses.

12.2.9.16 Activate Task.

SCI Command: AT

The Activate Task command is not processed by the Debugger. The command procedure issues an Activate Suspended Task SVC.

12.2.9.17 Halt Task.

SCI Command: HT

MODULE: D$$HT

CODE: >7

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the halt task command processor is D$$HT in module D$$HT. The function of D$$HT is to update the appropriate Debugger variables, to issue a Halt Task SVC for the specified task, and to display the debug panel.

In a way similar to the Activate Task command, the Halt Task command is not strictly associated with the Debugger. Any task in the user's job can be halted, regardless of whether or not it is the controlled task or whether or not there is debugging activity in process on any task.

The logic of D$$HT is shown in the following metacode:

```
D$$HT:
   Call D$PR1 to process the run ID;
   IF error
     THEN Return;
   Call D$HT to halt the task;
   Call D$PRP to build the debug panel and write it
     to the terminal;
```

12.2.9.18 Quit Debugger.

SCI Command: QD

MODULE: D$$QD

CODE: >10

PARMS:

PARM Number	Definition	Field Prompt Name
1	Whether to kill the task being debugged -- YES or NO	KILL TASK ?

The entry point for the Quit Debugger command processor is D$$QD in module D$$QD. The function of D$$QD is to reset all local variables with regard to the controlled task, and to dispose of the task, as requested. The logic of D$$QD is shown in the following metacode:

```
D$$QD:
   IF the controlled task ID is zero
     THEN report an error;
     ELSE Reset last 'FOR' value; clears simulation state information
           PC, WP, and status;
          Reset panel memory address to PC;
          Call D$DASB to delete all simulated breakpoints;
          Call D$DST to delete the symbol table;
          Reset the controlled flag in controlled task TSB;
          IF first PARM is YES
            THEN Issue Kill Task SVC on controlled task;
                 IF error
                   THEN Report it;
                 ENDIF;
          ENDIF;
          Reset DD$CTI, the controlled task ID;
   ENDIF;
```

Routine D$DST issues a Release Memory SVC to release the memory that was acquired for the symbol table. After that is done, the symbol table pointer, SYMTBL, is cleared.

12.2.9.19 Resume Task.

SCI Command: RT

MODULE: D$$RT

CODE: >08

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID

The entry point for the Resume Task command processor is D$$RT in module D$$RT. The function of D$$RT is to reactivate a task that is suspended because it executed a breakpoint sequence. The logic of D$$RT is shown in the following metacode:

```
D$$RT:
   Call D$PR1 to process the run ID;
   Call D$RTS to get state of task;
   IF task is suspended
     THEN Call D$PRC to proceed from suspension;
          Call D$DMC;
     ELSE Report an error;
   ENDIF;
```

12.2.9.20 Execute in Debug Mode.

SCI Command: XD

MODULE: D$$DEB

CODE: >0D

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of the task to be executed in debug mode	RUN ID
2	Pathname of symbol table file	SYMBOL TABLE OBJECT FILE
3	Machine code	990/12 OBJECT CODE?

The XD command processor is routine D$$DEB, in module D$$DEB. The function of D$$DEB is to establish the environment for a controlled task -- that is, a task that executes in debug mode.

The elements of the controlled task environment are as follows:

* The controlled bit in the TSB

* Last 'FOR', How many ST instructions were requested in last ST command

* Current context (WP, PC, and ST)

* 990/12 flag. Any value that starts with the character Y is interpreted to mean yes, and the flag is set.

Routine D$BST is called to initialize the task symbols table. This routine reads the symbol table produced by the assembler when the SYMT option is specified, and builds the internal table SYMTBL. The object code format is discussed in detail in the DNOS Assembly Language Programmer's Guide.

12.2.9.21 Execute and Halt Task.

The Execute and Halt Task command (XHT) is not processed by the Debugger. The SCI command procedure uses the .SVC primitive to bid the task with a call block structured to cause the operating system to halt the task immediately.

12.2.9.22 Find Byte.

SCI Command: FB

MODULE: D$$FB

CODE: >16

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID
2	List of integer values to find *	VALUE(S)
3	Starting memory address to search	STARTING ADDRESS
4	Ending memory address to search	ENDING ADDRESS

* The list of values must be enclosed in parentheses. The elements of the list are separated by commas.

The entry point for the find byte command processor is D$$FB in module D$$FB. The function of D$$FB is to scan each byte of the specified task until a byte or group of bytes matching the input value(s) is found. If starting and ending addresses are not specified, the entire task is scanned.

D$$FB calls D$BVT to read the list of values specified as the second element of the PARMS list, and builds a value table. D$BVT scans the list and converts each expression on the list to a 16-bit (Find Word) or 8-bit (Find Byte) value that is stored in the table. A maximum of 32 bytes of data can be stored in the value table against which the task is searched.

The list of values must be in parentheses. Each element in the list can be an expression. Comma is the delimiter between elements.

D$ESE is called to evaluate each item on the list.

NOTE

The logic in the command processors for Find Byte and Find Word makes the assumption that the two processors use the same workspace. They use unique labels in their transfer vectors (DW$$FB and DW$$FW, respectively), but those labels must identify the same workspace.

12.2.9.23 Find Word.

SCI Command: FW

MODULE: D$$FW

CODE: >17

PARMS:

PARM Number	Definition	Field Prompt Name
1	Run ID of task being debugged	RUN ID
2	List of HEX values to find *	VALUE(S)
3	Starting memory address to search	STARTING ADDRESS
4	Ending memory address to search	ENDING ADDRESS

* The list of values must be enclosed in parentheses. The elements of the list are separated by commas.

The entry point for the Find Word command processor is D$$FW in module D$$FW. The function of D$$FW is to scan each word of the specified task until a word or group of words matching the input value(s) is found. If starting and ending addresses are not specified, the entire task is scanned.

The list of values must be in parentheses. Each element in the list can be an expression. Comma is the delimiter.

NOTE

The logic in the command processors for Find Byte and Find Word makes the assumption that the two processors use the same workspace. They use unique labels in their transfer vectors (DW$$FB and DW$$FW, respectively), but those labels must identify the same workspace.

12.2.9.24 Assign Simulated Breakpoint.

SCI Command: ASB

MODULE: D$$ASB

CODE: >0C

PARMS:

PARM Number	Definition	Field Prompt Name
1	Event on which the breakpoint is to occur. The value must be one of the following: A (for memory alteration) C (for CRU address) P (for PC value) R (for memory reference) S (for status value) X (for XOP 15)*	ON (A,C,P,R,S,X)
2	Minimum address for the breakpoint	FROM
3	Maximum address for the breakpoint	THRU
4	Memory address to be displayed when breakpoint is reached	DISPLAY
5	Number of times breakpoint is to be encountered	COUNT

*Note that the X option is not allowed at the SCI command level.

The entry point to the Assign Simulated Breakpoint command processor is DP$$ASB in module D$$ASB. The function of D$$ASB is to enter the specified information in the simulated breakpoint table (EVENTS). The logic of D$$ASB is shown in the following metacode:

```
D$$ASB:
   IF no controlled task
     THEN Return;
   Find an unused entry in the breakpoint table;
   IF none found
     THEN Report error;
          Return;
   Determine breakpoint type;
   Make entries in the table entry;
   Generate a message to return breakpoint number to user;
   Call D$PRP to write the panel to display;
```

12.2.9.25 Delete Simulated Breakpoint(s).

SCI Command: DSB

OVERLAY: D$$DSB

CODE: >0E

PARMS:

PARM Number	Definition	Field Prompt Name
1-n	List of breakpoint numbers to be deleted	BREAKPOINT NUMBERS

The breakpoint numbers can be one of the following three types of input:

* A list of breakpoint numbers

* A character string that begins with the letters AL - which is interpreted to be a request to delete all breakpoints

* Null - which is processed to mean the breakpoint at the current value of the PC

The entry point for the delete simulated breakpoint command processor is D$$DSB in the module D$$DSB. The function of the processor is to remove one or more entries from the EVENTS table. The logic of D$$DSB is shown in the following metacode:

```
D$$DSB:
   IF no controlled task
     THEN Report error;
          Return;
   LOOP:  DO UNTIL (no more PARMS);
   Increment PARM counter;
   Call S$PARM to get next element on PARMS list;
   IF string is not null
     THEN IF string begins with AL
            THEN DO FOR (all breakpoint table entries);
                   Clear entry;
                 END DO;
                 Exit LOOP;
            ELSE Validate that breakpoint number is in
                 the proper range (0 through >FFFF);
                 IF Breakpoint is within table
                   THEN Make the entry null by clearing
                        first word;
                   ELSE Report the error and quit;
   END DO;
   Call D$PRP to display the debug panel;
```

Since the task is simulated, not executed, the address space of the task is not altered when a breakpoint is assigned, and need not be restored when the breakpoint is deleted.

12.2.9.26 List Simulated Breakpoints.

SCI Command: LSB

MODULE: D$$LSB

CODE: >0F

PARMS: None

The entry point for the List Simulated Breakpoints command processor is D$$LSB in module D$$LSB. The function of D$$LSB is to format and write to the interactive terminal (TLF) a summary of the information in the simulated breakpoint table, EVENTS. The logic of D$$LSB is shown in the following metacode:

```
D$$LSB:
   IF Controlled task ID = 0
     THEN Report an error;
          Return;
   Call S$OPEN to open the TLF;
   Call S$WRIT to write the breakpoint table header;
   DO UNTIL (No more table entries);
     IF table entry is non-null
       THEN Format breakpoint number (convert binary to ASCII);
            Call S$WRIT to write breakpoint number;
            Format breakpoint type;
            Call S$WRIT to write breakpoint type;
            DO for all positions in entry;
              Format information;
              Call S$WRIT to write information;
            END DO;
   END DO;
```

12.2.9.27 Resume Simulated Task.

SCI Command: RST

MODULE: D$$RS

CODE: >11

PARMS: None

The entry point for the resume simulated task command processor is D$$RS in module D$$RS. The function of D$$RS is to resume simulation of the controlled task after encountering a simulated breakpoint. The logic of D$$RS is shown in the following metacode:

```
D$$RS:
   IF no controlled task
     THEN  Report error and abort;
   IF current PC is not the same as the simulation PC
     THEN Update PC, WP, and ST;
   LABEL1:
     ELSE Initialize to execute one breakpoint;
          IF last suspend was user; default trap count
            THEN Set iteration count to one;
          ENDIF
          IF last suspend was breakpoint
             THEN Call D$DB to delete breakpoint;
                  Call D$STT to simulate one instruction;
                  Call D$AB to restore the breakpoint;
                  Adjust iteration count;
                  Inhibit breakpoints;  go over trap
   Set the iteration count;
   Call D$SMC;
   IF continue simulation
     THEN GO TO LABEL1;
```

D$SMC is the simulation mode controller for the RS and ST instructions. It simulates the controlled task to a breakpoint, formats and writes the breakpoint message to the screen, and accepts input from the terminal. D$SMC only processes the following event keys:

* CMD - Return to the caller

* Continue key - continue simulation request key

12.2.9.28 Simulate Task.

SCI Command: ST

MODULE: D$$S

CODE: >12

PARMS:

PARM Number	Definition	Field Prompt Name
1	Number of instruction simulations to be performed	FOR
2	Address of the first instruction to be simulated	FROM
3	Address of the last instruction to be simulated	TO

The entry point for the simulate task command processor is D$$S in module D$$S. The function of D$$S is to begin simulation of the controlled task. The logic of D$$S is shown in the following metacode:

```
D$$S:
   IF controlled task ID is zero
     THEN Abort with NO DEBUG TASK error code;
   Call D$PSP to process the second PARM;
   IF PARM is null
     THEN Assume current PC;
     ELSE IF not same value as simulation PC

   *  The task may have executed "RT" since the last simulation.
      This updates simulation PS, WP, and ST to match possible
      change.

           THEN Replace WP, ST and PC
   Call D$PSP to process the ending address PARM;
   IF ending address is null or zero
     THEN Use last good value (DD$LFR);

   *  Last valid 'FOR' count

   Call D$SMC to simulate task to next breakpoint;
   IF "continue simulation"

   *  Key returned (F3 key)

     THEN Call D$$RS to resume simulation;
   ENDIF;
```

12.2.9.29 Support Subroutines.

The subroutines discussed in this paragraph are called by many of the command processors. The function and logic of each of the following routines is discussed.

* D$PR1 and D$PR2 , two routines that process the run ID. They are designed with the assumption that they use the same workspace. They have unique labels used to declare the workspace, DW$PR1 and D$PR2, but they must be equal. The routines are functionally equivalent with the exception that D$PR2 tests for run ID value of S, which is changed to a run ID of zero. A run ID of zero means system memory instead of a task.

* D$HT halts the specified task.

* D$RST restores the status of a task.

* D$DMC is the debug controller. It controls either the simulation of the controlled task, or activates a task

that is not in debug mode.

* D$ESE evaluates an expression that may contain symbols.

* D$PRP examines the terminal mode and controls output of the debug panel.

* D$PSP writes the debug panel to the interactive terminal.

* D$POP builds and writes the debug panel to a VDT terminal.

* D$WL writes a line to the terminal.

The calling sequences for the subroutines are documented in the prologues of the code.

D$PR1 and D$PR2.

The function of D$PR1 and D$PR2 is to validate the run ID passed in the PARMS list. Both routines return an error code if the run ID is one of the following:

* Run ID = 0

* Run ID is for the Debugger itself

* Privilege level of the user is less than 2 and the run ID is for a system task.

D$PR2 contains one further test on run ID. If the run ID is the letter S, D$PR2 returns a value of 0 to the caller.

The entry point for D$PR1 is D$PR1 in module D$PR1. The entry point for D$PR2 is D$PR2 in module D$PR1.

D$HT.

The function of subroutine D$HT is to halt the specified task.

If the Run ID is zero, the task is not halted. Tasks in the following states are not halted:

* Waiting on diagnostic services

* Suspended for queue input. The only tasks that should be in state >24 are the nucleus function queue server system tasks. This exception is made to protect the integrity of the operating system.

D$HT has two entry points -- D$HT and D2$HT. The second entry point skips validation of the run ID and calling D$RTS to get the task state. The logic of D$HT is shown in the following metacode:

```
D$HT:
IF Run ID = 0
  THEN Take error exit;
Call D$RTS to get task state;
D2$HT:
Save original task state;
Eliminate the following cases from further processing;
      WAITING ON DIAGNOSTIC SERVICES
      SUSPENDED FOR QUEUE INPUT
Call D$RST to get FLAG2 word from TSB;
Save old FLAG2;
Issue SVC to suspend task;
OUTER LOOP: Set loop counter to MAX;
  INNER LOOP: Time delay 1/2 second;
              Get task state;
              IF state = suspended
                THEN Return;
              Decrement loop counter;
              IF loop count is zero
                THEN Exit INNER LOOP;
  END INNER LOOP;
            IF the task state is not suspended
              THEN Ask user whether to keep trying;
                  IF answer = YES
                    THEN GO TO OUTER LOOP;
                    ELSE
                       Call D$RST to get task state
                         Ask user if he wishes to execute command
                            without suspending the task;
                         IF answer = YES
                           THEN Return;
                           ELSE
                             Ask user whether to leave
                                 suspend pending;
                             IF answer = YES
                               THEN Set error code;
                                   Return;
                               ELSE
                                   Issue Activate
                                   Task SVC;
                                   Set error code;
                                   Return;
              ELSE Return;
END OUTER LOOP;
```

The value of MAX is 10 (approximately 5 seconds delay) in the current release. D$HT calls S$PKEY to handle I/O to the terminal when there is a need to communicate with the user.

D$RST.

The entry point is D$RST in module D$RST. The function of D$RST is to restore the previous state of the specified task. The logic is as follows:

```
D$RST:
   IF run ID = 0
     THEN Return;
   IF original state of task was suspended
     THEN Restore original TSB flag words;
          Return;
   IF original state of task was terminated memory resident
     THEN Call D$RTS to get current TSB;
          IF current state is terminated memory resident
            THEN Restore original TSB flag words;
            ELSE Return;
     ELSE Call D$PRC to restore TSB and activate task;
          Return;
   ENDIF;
```

D$DMC.

The entry point is D$DMC in module D$DMC. The function of D$DMC is to supervise the execution and simulation of the controlled task until it is in the unconditional suspend state, or until the user enters an event key. The logic of D$DMC is shown in the following metacode:

```
D$DMC:
   IF run ID is not the controlled task
     THEN Take error exit;
   Call D$WFT to wait for task suspension;

    D$WFT:DO UNTIL (task suspends) OR (event key);
             Call D$RTS to get task state;
             IF task state is suspended
               THEN IF halted by XOP 15, 15; bit set in TSB flag word
                      THEN Save the original TSB flag word
                             in DD$OF2;
                    Return;
             Issue Get Event Character SVC;
             IF event character input;
               THEN Halt the task;
                    Put event character in caller's workspace;
                    Return;
             Issue Time Delay SVC;
          END DO;
    END D$WFT;

   Call D$PIR to display data;
```

Regardless of which event occurs (task suspends or user enters an event key), the task is suspended when control returns from D$WFT to the calling routine.

D$PIR builds the display and writes it to the listing file. What is displayed depends on the mode in which the Debugger is operating, as follows:

* TTY - Display internal registers only

* VDT - When there is a controlled task, display the section of memory previously specified. If there is no section specified, D$PIR builds a display of memory beginning at the current PC.

<u>D$ESE</u>.

D$ESE evaluates symbolic expressions. The operators +, -, *, and / are allowed. Operands may be of the following types:

* Expressions consisting of constants and/or symbols.

* Indexed address of the form: Name(displacement)

* Indirect address of the form: <address>

where:

name, address, and displacement may be symbolic expressions.

Name is of the following form:

PHASE.IDT.SYMBOL

The following short forms are recognized:

* PHASE.IDT
* .IDT
* .IDT.SYMBOL
* ..SYMBOL

where IDT is the module identifier name

When PHASE is not specified, the previously specified PHASE is used. If no PHASE has been specified, 0 is the default. When IDT is not specified, the previously specified IDT is used.

The logic of D$ESE is shown in the following metacode:

```
D$ESE:
   IF first character is a period
     THEN Move past first character
          IF next character is a period
            THEN Process symbolic form ..NAME;
                 (Using default PHASE and IDT)
            ELSE Process symbolic form .IDT.NAME;
                 (Using default PHASE)
     ELSE Process symbolic form PHASE.IDY.NAME;
```

D$PRP.

The entry point for the D$PRP command processor is D$PRP in module D$PRP. The function of D$PRP is high-level control of processing the debug panel. The logic of D$PRP is shown in the following metacode:

```
D$PRP:
   Call S$STAT to get terminal status;
   IF TTY terminal or batch mode
     THEN Take error exit;
   Call D$POP to put out the panel;
```

The format of the debug panel is documented in the DNOS System Command Interpreter (SCI) Reference Manual, with the Show Panel (SP) command.

D$PSP.

The function of D$PSP is to process a symbolic expression that is an element of the PARMS list.

The logic of D$PSP is shown in the following metacode:

```
D$PSP:
   Call S$PARM to get the next element of the PARMS list;
   IF null parameter
     THEN Set error code;
     ELSE Call D$ESE to evaluate the expression and store
               it in the current workspace;
   ENDIF;
```

D$POP.

The entry point for D$POP is D$POP in module D$POP. The function of D$POP is to display the debug panel on a VDT terminal. The logic of D$POP is shown in the following metacode:

```
D$POP:
   Call D$CSR to clear screen;
   Format internal register line;
   Call D$WL to write workspace registers heading;
   Call D$RTS to get workspace pointer;
   Call D$DM to display memory;
   Call D$WL to write breakpoints heading;
   Call D$BBL to build breakpoints line;
   Call D$WL to write breakpoints line;
   Alter S$$MNU to preserve panel display;                *
   Call D$WL to write memory heading;
   IF memory display address is >FFFF
     THEN Replace address with current PC;
   Calculate number of bytes that can be displayed on
    remaining lines of screen;
   Call D$DM to display memory;
```

D$WL.

The entry point is D$WL in module D$WL. The function of D$WL is to write a formatted line to the terminal. The logic is shown in the following metacode:

```
D$WL:
   IF TTY
     THEN Call S$WIT to write carriage control;
   Adjust line number to row number (0 origin);
   Call S$WIT to write the text;
```

12.2.9.30 Pascal Debugging Commands.

The following SCI commands are used in debugging Pascal tasks. They are not documented in detail in this document

* Assign Breakpoint (Pascal)

* Delete Breakpoint (Pascal)

* Delete/Proceed from Breakpoint(s) (Pascal)
* List Breakpoints (Pascal)
* List Pascal Stack
* Proceed from Breakpoint (Pascal)
* Show Pascal Stack

12.2.10 Modifying the Debugger.

Since the driver is table driven, to add a new function, add an overlay to the overlay table and an entry point to the command processor address table.

12.2.10.1 Changing the code.

The logic in the command processors for Find Byte and Find Word makes the assumption that the two processors use the same workspace. They use unique labels in their transfer vectors (DW$$FB and DW$$FW, respectively), but those labels must point to the same workspace.

12.2.10.2 Maintenance.

The logic of D$OV1 is built on the assumption that the List Memory command has a zero value.

The Debugger is not necessarily compatible with SCI when interfacing with the user. The number of event characters processed is limited, and the only function is that of inputting data.

12.2.11 Internationalization.

All messages displayed to the user are defined in module D$MSG. Translating the text of these messages makes the Debugger speak a language other than English.

12.3 LLR

LLR is an assembly language program that formats and writes one or more logical records of a file to a specified listing file. The code that is unique to LLR is a single module named LLR, in the source directory.

12.3.1 Structure of the task.

LLR is a replicatable task segment that contains the unique LLR code, the O$ library of I/O routines, and modules from the UTCOMN library.

The O$ routines are documented in the section of this manual entitled File Maintenance Utilities. This collection of routines does some I/O processing and interfaces with S$SYSTEM I/O routines in the shared procedure segment.

The UTCOMN routines are documented in the section of this manual entitled Conventions and Libraries.

LLR requires access to routines in the S$SYSTEM shared procedure segment.

12.3.2 Coding Conventions.

DNOS coding conventions are not followed in LLR. Data structure names are descriptive. Address labels within the code are generally two alphabetic characters followed by two numeric characters.

12.3.3 Flow of control.

LLR is bid by SCI during processing of the LLR command. LLR writes the requested records to the specified file and terminates by calling either STOPS or FINISH, common exit routines in the UTEXIT module of the UTCOMN library.

12.3.3.1 Invoking LLR.

When SCI bids the LLR task, no CODE value is passed to the task. The following information is passed in the PARMS list on the bid statement:

PARM Number	Definition	Field Prompt Name
1*	1*	None
2	Pathname of the file to be listed	PATHNAME
3	First record to be listed	STARTING RECORD
4	Number of records to list	NUMBER OF RECORDS
5	Output file for listing	LISTING ACCESS NAME
6	Logical record size	None
7	Open rewind?	None

* The first element in the PARMS list is not currently used. It is defined for historical reasons only.

NOTE

Refer to the DNOS System Command Interpreter (SCI) Reference Manual for further information on the field prompts.

The command procedure LLR in the current release of DNOS does not specify the Open rewind? PARMS. A default of YES is used in all cases.

12.3.3.2 Initialization.

Initialization for LLR is done at the beginning of the module, as well as through DATA statements at the end of the module.

12.3.3.3 Subroutine Linkage.

LLR uses the UTCOMN routines UTPUSH and UTPOP to stack and unstack subroutine calls.

12.3.3.4 Error Processing.

LLR calls the UTCOMN routine STOPS to process nonrecoverable errors. The nonrecoverable errors are as follows:

* Error from incorrect elements in the PARMS list

* Error from S$ISUB when calculating the number of records to space forward in input file

* Error from GTNXT (get next record number)

* Error from S$ISUB when calculating the number of records to list

* Error from SVC to process the input and output files.

12.3.3.5 Termination.

When all records are written to the listing file, LLR calls the UTCOMN routine FINISH to terminate. When an nonrecoverable error occurs, LLR terminates by calling the UTCOMN routine STOPS.

12.3.4 Data Structures and Files.

The following data structures for LLR are defined at the end of the source code module:

* SVC call blocks and opcodes

* Buffers for input parameters

* Constants and temporary buffers for 32-bit arithmetic

* SYSDAT, the call block to access the NFDATA common segment for the country code.

* Country code constants

* 8-bit ASCII flag bytes

* Text strings for the headings to be written to the output file. Equates are used at the end of each text string so that a label is defined. The names of the data structures and their English contents are as follows:

Name	Contents
RECMSG	RECORD:
TTLMSG	FILE ACCESS NAME:
SAMBUF	SAME

LLR has no files that are specifically its own. The input file specified in the PARMS list is opened for read access only. LLR optionally rewinds the output file, depending on the seventh element of the PARMS list, then writes the listing to it.

12.3.5 Detailed Design.

The logic of LLR is shown in the following metacode:

```
Call O$INIT to initialize the O$ routines environment;
Call O$PARM to get first element of PARMS list;
IF pathname is null
  THEN Direct output to the TLF;
Call O$SOUT to initialize for output;
Call O$PARM to get second element of PARMS list;
IF beginning line number is not specified
  THEN Substitute -1 to show none specified;
Call O$PARM to get third element of PARMS list;
IF ending line number is not specified
  THEN Substitute -1 to show none specified;
Calculate number of records;
IF logical record length is not specified
  THEN Use 512;
IF rewind parameter is not specified
  THEN Set the flag to indicate YES;
Issue Assign LUNO SVC for input file;
Get Country Code from NFDATA;
IF Japanese or Arabic
  THEN Set proper bits for use of 8-bit character codes;
Issue Open File SVC for input file;
Issue Get Memory SVC for logical record length;
Position input file to specified beginning record;
DO UNTIL (EOF) OR (Last record requested);
  Read a record;
  Write header line;
  Write record number;
  Format EOL and EOR cases (blank fill short buffer);
  Call O$LINE to print the record;
END DO;
Terminate;
```

When positioning the input file to the beginning record, if the record number is greater than 65,536 (16-bits), the SVC must be issued twice because the call block has a one-word data field.

12.3.6 Internationalization.

All labels and headings displayed to the user are in the data structure declarations at the end of module LLR.

12.4 MRFSRF

MRFSRF modifies or shows the contents at a specified offset in a record of a file. MRFSRF is written in assembly language. The code unique to MRFSRF is in a single module named MRFSRF in the source directory for MRF.

12.4.1 Structure of the task.

MRFSRF is a replicatable, software-privileged task. It must be software-privileged in order to issue the SVC to open the file unblocked. The MRFSRF task segment contains the unique MRF code, the O$ library of I/O routines, and modules from the UTCOMN library. The O$ routines are documented in the section of this manual entitled File Maintenance Utilities. This collection of routines does some I/O processing and interfaces with S$SYSTEM I/O routines in the shared procedure segment.

The UTCOMN routines are documented in the section of this manual entitled Conventions and Libraries.

MRFSRF requires access to routines in the S$SYSTEM shared procedure segment.

12.4.2 Coding Conventions.

DNOS coding conventions are not followed in MRFSRF. Data structure names are descriptive. Address labels within the code are, generally, two alphabetic characters followed by two or more numeric characters.

In the declarations, variables that will be known at run time are initialized to zero with the following DATA statement:

```
label DATA $-$
```

12.4.3 Flow of Control.

MRFSRF is bid by SCI. The task processes the request, writes the appropriate display, and then terminates.

12.4.3.1 Invoking MRFSRF.

MRFSRF is bid by SCI during processing of the Modify Relative to File (MRF) command procedure and the Show Relative to File (SRF) command procedure. No CODE value is passed. The PARMS list is as follows:

PARM Number	Definition	Field Prompt Name
1	Request code >42: Modify >41: Show	None
2	Input file pathname	PATHNAME
3	Record number	RECORD NUMBER
4	Offset to first word	FIRST WORD

The remaining elements are not the same for the two commands.

MRF PARMS List

5	List of replacement values	DATA
6	List of verification values	VERIFICATION DATA
7	File to which processing messages are written	OUTPUT ACCESS NAME
8	Checksum value	CHECKSUM

SRF PARMS List

5	Address of last word to be shown	None
6	File to which processing messages are written	OUTPUT ACCESS NAME

NOTE

Refer to the DNOS System Command Interpreter (SCI) Reference Manual for further information on the field prompts.

12.4.3.2 Initialization.

Initialization for I/O is done in the O$ routine O$INIT.

12.4.3.3 Major Loop/Routines.

The first part of the code in module MFRSRF is common for both the modify and show operations. Following the common code, control is transferred to either the SHOW processor or to the MODIFY processor.

12.4.3.4 Error Processing.

MRFSRF calls the UTCOMN routine STOPS to process nonrecoverable errors. The nonrecoverable errors are as follows:

* Unable to assign LUNO to input file
* Attempt to modify data beginning at an address that is not even
* Error in attempting to read a record from the file
* Error from S$ISUB or S$IADD when calculating a record number

12.4.3.5 Termination.

MRFSRF terminates by calling UTCOMN routine FINISH when there is no error to report, or by calling UTCOMN routine STOPS when there is an error to report.

12.4.4 Subroutine Linkage.

The UTCOMN routines in modules UTPUSH and UTPOP are called to stack and unstack branch and link/return subroutine calls. STACK is the local data structure used for register storage and retrieval.

12.4.5 Data Structures and Files.

Data structures are specified at the end of the module MRFSRF. The structures can be grouped into the following categories:

* A single sixteen-register workspace (WSP).
* Buffers for input parameters. This includes a 100 byte buffer for manipulation of parameters that are lists.
* Text strings for all headings written to the output file
* SVC call blocks, including the call block to retrieve the country code from NFDATA.
* Constants and temporary buffers for 32-bit arithmetic. MRFSRF is designed to process files with record numbers as large as 32 bits.
* Constants related to manipulation of country code and 8-

bit ASCII requirements

MRFSRF uses two files, the input file and the listing file. The input file is is opened by issuing an SVC with the specification that the open be unblocked so that any type file can be treated as a relative record file. For the SRF command, read only access is requested. For the MRF command, exclusive write access is required.

If no listing file is specified, all output is written to the TLF.

12.4.6 Detailed Design.

MRFSRF is divided into the following three parts:

* Common code that performs common processing for the show and modify commands

* Code that is specifically show functions

* Code that is specifically modify functions

The logic of MRFSRF is shown in the following metacode:

```
Obtain common parameters:
  Parameter type, pathname, record number, first word;
Assign task-local LUNO to input file;
Open input file as a relative record file;
Calculate the record needed;
Issue a Get Memory SVC to obtain space for the record;
Get and test country code for 8-bit ASCII;
Read the record into the buffer;
IF parameter type is modify
  THEN Verify input data, if requested;
       Load new data into buffer;
       Write buffer back to disk;
       IF not all of data
         THEN Load the rest;
       Write record back to disk;
  ELSE Initialize output file;
       Write header line;
       Write address;
       Write eight words and corresponding ASCII;
       Delete duplicate lines;
END;
```

12.4.7 Internationalization.

All text strings written to the output file are defined in the data structures section at the end of the code module. The names of the data structures and their contents (in English) are as follows:

Name	Contents
VERHDR	VERIFICATION DATA
ORGHDR	CURRENT DATA
NEWHDR	NEW DATA : CHECKSUM=
RECMSG	RECORD:
TTLMSG	FILE:
SAMBUF	SAME
HDRCHK	CHECKSUM:
CUMHDR	CUMULATIVE CHECKSUM:

12.5 MPISPI

MPISPI modifies or shows the contents of a program image in a specified file. The program image may be a procedure segment, a task segment or an overlay segment. When a program file is modified, the relocation bit map may be optionally updated. MPISPI is written in assembly language. The code unique to MPISPI is in a single module named MPISPI in the source directory MPI.

12.5.1 Structure of the task.

MPISPI is a replicatable, software-privileged task. It must be privileged to write to a program file. The MPISPI task contains the transfer vector, the unique MPI code, the O$ library of I/O routines, and modules from the UTCOMN library.

The O$ routines are documented in the section of this manual entitled File Maintenance Utilities. This collection of routines does some pre-processing and interfaces with S$SYSTEM I/O routines in the shared procedure segment.

The UTCOMN routines are documented in the section of this manual entitled Conventions and Libraries.

MPISPI requires access to routines in the S$SYSTEM shared procedure segment.

12.5.2 Coding Conventions.

DNOS coding conventions are not followed in MPISPI. Data structure names are descriptive. Address labels within the code are, generally, two alphabetic characters followed by four numeric characters.

The entry point for each subroutine is two alphabetic characters followed by four zeros.

12.5.3 Flow of Control.

MPISPI is bid by SCI. The task processes the request, writes the results to the listing file, and then terminates. MPISPI does not interface with the interactive terminal.

12.5.3.1 Invoking MPISPI.

MPISPI is bid by SCI during processing of the Modify Program Image (MPI) command procedure and the Show Program Image (SPI) command procedure. No CODE value is passed. The PARMS list is as follows:

PARM Number	Definition	Field Prompt Name
1	Request code >43: Modify >44: Show	None
2	Program file pathname	PROGRAM FILE
3	Pathname of listing file	OUTPUT ACCESS NAME
4	Segment type (only two characters are examined by the program) TA: Task PR: Procedure OV: Overlay SE: Segment	MODULE TYPE
5	Segment identification	MODULE NAME OR ID
6	Hexadecimal address	ADDRESS

The remaining elements are not the same for the two commands.

MPI PARMS List

7	List of verification values	VERIFICATION DATA
8	List of replacement values	DATA
9	Checksum value	CHECKSUM
10	Whether or not to update the relocation bit map	RELOCATION OF DATA?

SPI PARMS List

7	Number of bytes of data to show	LENGTH

NOTE

Refer to the DNOS System Command Interpreter (SCI) Reference Manual for further information on the field prompts.

12.5.3.2 Initialization.

Initialization for I/O is done in the O$ routine O$INIT. MPISPI local variables are initialized at the beginning of the module, and through the use of DATA statements where the variables are declared.

12.5.3.3 Major Loop/Routines.

The first part of the code in module MPISPI is common for both the modify and show operations. Following the common code, control is transferred to the appropriate processor.

12.5.3.4 Error Processing.

MPISPI calls the UTCOMN routine UTSERR to report an SVC error and then return to SCI. MPISPI does not invoke the return code processor for an SVC error.

The UTCOMN routine UTUERR is called to report an other error, and then return to SCI. MPISPI makes no attempt to recover from the following errors:

* Error in attempting to access the communication area (TCA)

* Error in opening or closing the program file

* Error in accessing the specified segment in the program file

* Error in processing PARMS list

* Error in attempting to read or write to the program file

* Error in displaying the program image

* Error in processing the verification data. There is one case in which the mismatch of verification data and the program file is not considered an error -- when the replacement data matches the data in the program file.

* SVC error

MPISPI never returns any variable text.

12.5.4 Termination.

MPISPI terminates by calling UTCOMN routine UTSERR when there is an SVC error to report. Otherwise, MPISPI calls UTUERR with a condition code to reflect whether or not an error is to be reported.

12.5.5 Subroutine Linkage.

The UTCOMN routines in modules UTPUSH and UTPOP are called to stack and unstack branch and link subroutine calls.

12.5.6 Data Structures and Files.

MPISPI is designed to process files with record numbers as large as 32 bits. The current record number is kept in a two-word data structure. The two halves are at labels RCORD1 (high order bits) and RCORD2 (low order bits).

The following variables contain information about the segment currently being processed:

* RECLEN - the record length of the program file
* BASE - load address of the segment
* SRCRD - starting record number for the segment
* MODLEN - length of the segment, in bytes

RECBUF is a 512-byte buffer for program file records. This size accommodates two file overhead records.

Control flags:

* OPEN - whether or not the program file is successfully opened
* RMOD - whether or not the current record has been successfully modified

The following variables are MPISPI local:

* SVCERR - SVC error call block address
* S$ERR - Termination status
* CADDRS - Current word address
* CLEN - Offset from the beginning address displayed and CADDRS.
* DATA - a three-byte structure for storing the ASCII/JISCII representation (or a period, if the number is not a displayable character) of the contents of a word of memory, followed by a space.

The DX10 terminology PRB is used to identify IRBs. The following data structures are defined:

* SYSDAT - Call block for accessing the NFDATA variable country code

* PRB - I/O call block

WD3158 is a one-word constant with the hexadecimal value >3158. >31 is the Map Name to ID SVC opcode and >58 is the error code for a module that does not exist. After the SVC is issued, WD3158 is used as the comparison value to test the call block to determine whether or not the SVC was successful.

NIDFLG is a table of starting record numbers for the program file directory. It reflects a fixed organization of the program file.

The following constants are defined to describe the program file segment:

* DIRLEN - Length (in bytes) of the segment

* DIRFLG - Flags that indicate whether or not the segment is in use

* DIRREC - Record number at which the segment begins

* DIRLOD - load address of the segment

NOTE

Program files are always in relative record file format. Refer to the DNOS System Design Document for further information on organization of relative record files. They are discussed in the section entitled Disk Structures and File I/O.

12.5.7 Detailed Design.

The logic of MPISPI is shown in the following metacode:

```
Initialize subroutine linkage stack;
Call O$INIT to initialize O$ routine environment;
Initialize internal control variables;
Get country code;
Set 8-bit ASCII and display flags;
Call PR0000 to process PARMS list and open files;
Call LM0000 to locate the module in the program file;
IF request code is MODIFY
  THEN Call VR0000 to check the verification data;
       Call MD0000 to modify the program image;
Call DS0000 to display the program image;
IF request code is SHOW
  THEN Construct CHECKSUM header line;
       Write out the line;
Close the listing file;
IF program file was not opened
  THEN IF last record was not modified
         THEN Write out last record;
Close program file;
IF SVC error occurred
  THEN Set condition code indicator for SVC error;
       Call UTSERR;
  ELSE Set condition code indicator for utility results;
       Call UTUERR;
END;
```

The local subroutines prologues contain details of each subroutine. The subroutine entry points and their major functions are as follows:

PR0000 Fetches and processes the PARMS list. The processing includes translating the first two characters of the the segment type in the PARMS list to an integer as follows:

TA:	2
OV:	1
Other:	0

If the number of bytes to display is not specified with the show request code, the default of >10 is used.

LM0000 Locates the directory entry for the specified segment and saves the base address (load point), the record number in which the segment begins, and the length of the segment.

VR0000 Processes the verify data for a modify command. It returns one of the three following results:

The verification data matches the contents of the program file. No error code is set, and processing in MPISPI continues.

The verification data does not match the contents of the program file, but the replacement data matches the contents of the program file. In this case, an error code that results in an informative message is returned, and processing in MPISPI continues.

The verification data does not match the contents of the program file, and neither does the replacement data. This is an error that requires MPISPI to abort processing of the modify request.

MD0000	Modifies the program file, and if requested, updates the relocation bit map.

NOTE

Refer to the DNOS System Design Document for additional information on relocation bit maps. The primary reason the capability is included in the design of MPISPI is for the support of other operating systems that relocate overlays at run time.

DS0000	Formats and displays data. The data is displayed in both ASCII and hexadecimal.
LN0000	Formats and writes one line of the display.
NI0000	Issues the appropriate SVCs to maps the name of the segment to an ID in the program file.
GT0000	Gets a word from either the verification data buffer or the program file record buffer.
ST0000	An alternate entry point to GT0000, which writes data rather than reading it.
RR0000	Reads a record from the program file. If the record currently in the buffer has been modified, it is written to the program file
WT0000	Writes the current record to the program file.

12.5.8 Internationalization.

All text strings written to the output file are defined in the data structures section at the end of the code module. The names of the data structures and their contents (in English) are as follows:

Name	Contents
HDRVRD	VERIFICATION DATA
HDRIMD	CURRENT IMAGE
HDRNIM	NEW IMAGE: CHECKSUM =
HDRCHK	CHECKSUM =
SAME	SAME

The following fixed character strings are used in PR0000 to translate segment type to integer:

* TA
* OV
* PR
* SE

SECTION 13

VOLUME UTILITIES

13.1 INTRODUCTION

This section includes details on several of the utilities that handle disk volumes. The Copy Volume (CV) and Backup Directory to Device (BDD) processors share a common user interface. That interface is described here, along with the design of CV and BDD. These utilities can be used for either DNOS or DX10 disks. Therefore, some file names mentioned here are relevant to DX10 and others to DNOS.

This section also includes a description of the algorithm used to analyze a disk surface. This algorithm is part of the Initialize Disk Surface (IDS) command and part of the tape build process.

13.2 CVINIT -- PREPROCESSOR TASK FOR CV AND BDD

CVINIT provides the interface between the user and the Copy Volume (CV) and Backup Directory to Device (BDD) utilities. It is bid from either the CV or BDD command procedures. CVINIT performs the same functions for CV as it does for BDD except where noted. CVINIT accepts and validates input from the user, saves this data in a file on the system disk, assigns LUNOs, opens appropriate devices, and quiets the system if the system disk drive is to be used. CV, BDD, and CVINIT provide a rerun capability so that a series of copies can be made with all user input occurring at the start of the entire process.

13.2.1 CVINIT Data Definitions and Structures.

13.2.1.1 PARM_ARRAY.

The primary data structure of CVINIT is PARM_ARRAY. It is used to hold all of the information entered by the user for a given copy or backup. It is defined as follows:

```
TYPE
   PROMPT_CHARS      = PACKED ARRAY[1..PROMPT_LENGTH] OF CHAR;

   PARM_VAL          = RECORD
                          SIZE : BYTE;
                          VALUE: PROMPT_CHARS;
                        END;

COMMON
   PARM_ARRAY        : ARRAY[1..NUM_PARMS] OF PARM_VAL;
```

Each element of PARM_ARRAY contains a response to a prompt and the length of that response.

13.2.1.2 Saved Data File.

The data collected by CVINIT is stored on the system disk in one of two files. For Copy Volume, data is stored in .S$CV. For Backup Directory to Device, data is stored in .S$BDD. The data for all copies to be made are put in the appropriate file. These files are not deleted when the copy completes.

The file .S$CV has the following form:

```
All but last record:
                   <src. luno><dest. luno><list luno>
                   <interactive luno><b><src. device>
                   <b><dest.device><b><list device>
                   <b><src. volume><b><dest. volume><b><verify?>
                   <b><more copies?><b><conv. seq.>
                   <b><conv. rel-rec><b><run #>

Last record:       <flags><interactive device name>
```

Example:

```
FILE ACCESS NAME:   S$CV
RECORD:  000000
0000  0405 0200 0444 5330 3204 4453 3036 0444    .. .. .D SO 2. DS 06 .D
0010  554D 5906 4441 5441 3241 0644 4154 4132    UM Y. DA TA 2A .D AT A2
0020  4203 5945 5303 5945 5302 4E4F 0359 4553    B. YE S. YE S. NO .Y ES
0030  0131 0000 0000 0000 0000 0000 0000 0000    .1 .. .. .. .. .. .. ..
  SAME
005E  0000                                       ..
RECORD:  000001
0000  0708 0300 0444 5330 3204 4453 3036 0444    .. .. .D SO 2. DS 06 .D
0010  554D 5906 4441 5441 3541 0644 4154 4135    UM Y. DA TA 5A .D AT A5
0020  4203 5945 5302 4E4F 024E 4F02 4E4F 0132    B. YE S. NO .N O. NO .2
0030  0000 0000 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
005E  0000                                       ..
RECORD:  000002
0000  0000 5354 3038 0000 0000 0000 0000 0000    .. ST 08 .. .. .. .. ..
  SAME
005E  0000                                       ..
```

The flags word has the following form:

Bit 0	:	Set to 1 if the system disk drive is involved.
Bits 2-15	:	Reserved for future use.

The file .S$BDD has the following form:

All but last record:	<src. luno><dest. luno><list luno> <interactive luno><b><src. device> <b><dest. device><b><list device> <b><src. path name><b><dest. pathname> <b><verify?><b><more backups?> <0><0><b><run #>
Last record:	<flags><interactive device name>

Example:

```
   FILE ACCESS NAME:  .S$BDD
   RECORD:  000000
   0000  0A0B 0600 0444 5330 3204 4D54 3031 0444    .. .. .D SO 2. MT 01 .D
   0010  554D 5914 4441 5441 322E 5745 454B 4C59    UM Y. DA TA 2. WE EK LY
   0020  2E52 4550 4F52 5453 0003 5945 5303 5945    .R EP OR TS .. YE S. YE
   0030  5300 0001 3200 0000 0000 0000 0000 0000    S. .. 2. .. .. .. .. ..
     SAME
   005E  0000                                       ..
   RECORD:  000001
   0000  0D0E 0900 0444 5330 3204 4453 3036 0444    .. .. .D SO 2. DS 06 .D
   0010  554D 590C 4441 5441 322E 414E 4E55 414C    UM Y. DA TA 2. AN NU AL
   0020  0942 4B55 502E 4649 4C45 0359 4553 024E    .B KU P. FI LE .Y ES .N
   0030  4F00 0001 3300 0000 0000 0000 0000 0000    O. .. 3. .. .. .. .. ..
     SAME
   005E  0000                                       ..
   RECORD:  000002
   0000  0000 5354 3038 0000 0000 0000 0000 0000    .. ST 08 .. .. .. .. ..
     SAME
   005E  0000                                       ..
```

The flags word has the following form:

```
     Bit 0      :  Set to 1 if the system disk drive is involved.
     Bits 2-15  :  Reserved for future use.
```

13.2.1.3 .S$CVI - The CVINIT Temporary File.

This file is used by CVINIT to keep track of whether the system disk will be used in any of the requested copies. It consists of one word of flags. The only currently defined flag is bit 0. If this bit is on, the system disk will be involved in at least one of the copies. On the first bid of CVINIT, the data is always written to .S$CVI. If the system disk is not involved in the first copy, a word of zeros is written. Otherwise, a word in which only bit 0 is set is written. On all subsequent bids of CVINIT, if there are any, data is only written to .S$CVI if the system disk is involved in that particular copy. On the last bid of CVINIT, .S$CVI is read and the contents of its first word are written to the last record of the saved data file. This serves to alert CV or BDD that the system disk drive will be involved in at least one copy.

13.2.2 CVINIT Algorithm.

The main program is named CVINIT and it drives the flow of control. It performs the following functions:

1. Perform self-identification. (CVISLF)

2. Check device type of interactive device and set VDT

flag. This is used by the message routines to determine if VDT or TTY I/O should be done.

3. Get all parameters supplied by the user. These are put in PARM_ARRAY. (CVIGET)

4. Open the listing device after mapping its synonym. (CVISYN, CVIOPN)

5. For the first run of the utility, write a header message to the listing device. (CVIHDR)

6. Print the user's input parameters on the listing device. (CVIPRT)

7. Open the saved data file. (CVISDF)

8. Verify that the user's input parameters are correct. (CVIVER)

9. Write COM_FLAGS to the temporary file, .S$CVI, to keep track of system disk drive involvement.

10. If the system disk drive is involved in the copy or backup, make sure that no other activity is going on. (CVIQUI)

11. If this is the last copy or backup requested by the user, do the following:

    a. Unload the disks that are involved. (CVIUNL)

    b. Read .S$CVI to find out if the system disk drive was involved in any of the copies. This will be written to the last record of the saved data file.

    c. Write contents of PARM_ARRAY (input parameters and LUNOs) to the saved data file. (CVIWRT)

    d. Write the last record of the saved data file. (Contains flags word and interactive device name.)

    e. Bid the copy or backup task and terminate. (CVIBID)

12. If this is not the last copy or backup requested by the user, do the following:

    a. Unload disks that are involved. (CVIUNL)

    b. Write the contents of PARM_ARRAY to the saved data file. (CVIWRT)

c. Terminate and return to the BDD command procedure. (R$TERM)

13.2.3 CVINIT Module Descriptions.

All modules are coded in Pascal except where noted. Also, any differences in execution between operating systems are noted. The source modules for CVINIT may be found in the following directories:

Pascal Source	DSC.DP.CV.PSOURCE
Assembler Source	DSC.DP.CV.SOURCE

All CVINIT source modules are named CVIxxx where xxx is a unique mnemonic identifier of the module.

13.2.3.1 CVIBID.

This routine bids the CV or BDD task off the utility program file. The task to bid is determined by the global variable WHICH_TASK, which is set in CVIGET. CVIBID passes the size, in beets, of the saved data file to the task it bids.

13.2.3.2 CVICLS.

This routine closes a device or file and release its LUNO.

13.2.3.3 CVIDEV.

This routine performs a verify device name SVC on an access name.

13.2.3.4 CVIERR.

This routine is a general error handler. All errors returned by an SVC or from an R$ routine pass through CVIERR. The input parameters contain the completion code, the address of any variable text associated with the error, the error source flag (denotes which DNOS expanded message file to use) and the error number or pointer to the SVC call block. Errors returned from an SVC are handled differently than other errors. The address of the offending SVC call block is passed to Return Code Processor SVC (>4C). This formats error message data in a manner acceptable to R$CMSG. R$CMSG is then called to get the error message. CVIMSG is called to write the message to the interactive and listing devices. This routine will write an error message no longer than two lines. It then terminates by a call to S$TERM. Rather than simply passing the error to R$TERM, R$CMSG is used and the message is written in order to display the error message between reruns. If this is the last rerun, the synonym $MC (which corresponds to the MORE COPIES? prompt) is set to YES. This forces the command procedure to display another set of prompts to give the user another chance.

13.2.3.5 CVIFIL.

This routine writes a record to the saved data file. The address of the buffer containing the record to be written, the length of that buffer, and the record number to write to are all input parameters.

13.2.3.6 CVIGET.

This routine retrieves the user's input from SCI via the R$PARM function. They are put into PARM_ARRAY in the following form:

```
<byte count, parm string>
<byte count, parm string>
```

The order in which they are put into the array is determined by a series of constants defined in the CONST template. These constants are used as array indices. If the user requests that more copies or backups are to be made, the common variable MORE_COPIES is set to true. The user makes this request by answering the MORE COPIES? or MORE BACKUPS? prompt from the command procedure. If the maximum number of reruns is exceeded, MORE_COPIES is set to false and no more SCI prompts will be offered to the user. The number of the current run is converted to an integer and will be used to select the record number in the saved data file to which information will be written. The type of copy requested (backup or copy) is determined by a check for a unique character sequence, which is passed by the command procedure to CVINIT. The common variable WHICH_TASK is given the value of this type of copy. If a backup is to be done, the source pathname undergoes synonym and logical name mapping since a synonym or logical name is allowed for that prompt. Next, CVIGET puts a copy of the system disk PDT in a buffer to be used later. If a backup was requested, the source pathname is checked for an implied system disk. If the system disk drive is involved in this copy or backup, a flag is set in the common variable COM_FLAGS (this flag is bit 0).

13.2.3.7 CVIHDR.

This routine writes the date and time, along with a header message, to the listing device.

13.2.3.8 CVIIMP.

This routine inserts the volume name of the system disk in front of the period in the source pathname field of PARM_ARRAY. This is only done for a backup operation (BDD).

13.2.3.9 CVILJB.

This assembly language routine disables the system log and DIOU, and sets the job limit to two (only this job and the system job). This is done by putting a zero in the task ID fields of the system log and DIOU queues, and setting the job limit field in NFJOBC to 2. The old values of the modified data are stored in common variables. It also disables crash dumps to prevent damaging a data disk. This module is only called when the system disk drive is involved.

13.2.3.10 CVILUN.

This routine assigns LUNOs and opens source and destination devices. It puts these values in the output parameter LUNOs. If there are no more copies or backups to be made, it also puts the interactive device LUNO in this parameter.

13.2.3.11 CVIMAP.

This routine gets the value of a logical name. The name is pointed to by an input parameter. It first looks for a local name. If one is found, return a pointer to the name to the calling routine. Otherwise look for a global name. If found, return a pointer to the value. If a name is not found return the Get Name's Value SVC call block to the calling routine.

13.2.3.12 CVIMSG.

This procedure writes messages to the user's terminal and reads the responses (if any). Messages and responses are also written to the listing device. If the user specified his terminal as the listing device (not recommended, but possible), only write the message once. It initiates a write of the message to the terminal. If a reply is needed (specified by an input parameter of Y), this will be a write with reply. The routine writes the message to the listing device if it is different from the terminal. If a reply was requested, a test is made for valid responses (only Y and N are valid). The routine keeps asking until a valid response is given. The reply is passed back to the calling routine. The second parameter does double duty. On input it specifies whether or not a reply is needed (Y = reply, N = no reply). On output it is the value of the reply (Y or N).

13.2.3.13 CVINAM.

This routine passes over PARM_ARRAY and calls CVIMAP in an attempt to map a user's device name inputs to their logical names' values (if any). Each name's value replaces the name in PARM_ARRAY.

13.2.3.14 CVINJB.

This routine reads the NFJOBC data structure and returns the number of jobs in the system to the calling routine.

13.2.3.15 CVINIT.

This is the main program and serves as the driver routine.

13.2.3.16 CVIOPN.

This routine assigns a LUNO to a device or file and opens it. If there is an error from the assign LUNO, it checks to see if it is a logical name and retries the assign LUNO.

13.2.3.17 CVIPAT.

This is patch area.

13.2.3.18 CVIPDT.

This routine searches the PDT chain for the system disk PDT. This is denoted by the system disk flag in the PDT.

13.2.3.19 CVIPRT.

This routine prints the SCI prompts and the user's responses to the listing device in the format that the user sees on the screen, that is, SCI-like format as well as the command procedure header. The user's responses are kept in PARM_ARRAY. Unfortunately, the order of these responses in PARM_ARRAY is not the same order that the user saw on his SCI screen. Therefore, a CASE statement is used to pluck the responses from PARM_ARRAY in the order familiar to the user. The case statement is switched on a value called LNUM, which is also the index of a FOR loop. The FOR loop loops on a series of constants defined in the constant template. Each of these constants is the index of a particular user response in PARM_ARRAY. Since the prompts for a backup are different from those for a copy, there are two FOR-loop/CASE constructs, one for backup and one for copy.

13.2.3.20 CVIQUI.

This routine makes sure that there is no other activity in the system. This means that no other jobs are active. If the system is not quiet (as determined by CVINJB), a message is sent to the user requesting that the system be quieted, and asking if they are ready. If they are ready, it makes sure the system is quiet and calls CVILJB to disable other activity from occurring. If the user responds that they are not ready, it asks if they want to quit. If they do not want to quit, it requests that the system be quieted. If they want to quit, it terminates normally and informs the user that termination was requested by the user.

13.2.3.21 CVISDF.

If this is the first backup or copy, this routine deletes any existing saved data file. It recreates and opens the file.

13.2.3.22 CVISLF.

This routine gets the station ID of the user's terminal, converts it to ASCII, and passes it back to the calling routine in the form STxx.

13.2.3.23 CVISYN.

This routine calls R$MAPS to map the value of a synonym. It replaces the synonym with its value in PARM_ARRAY.

13.2.3.24 CVISYS.

This routine obtains the NFDATA system data structure address.

13.2.3.25 CVITRM.

This termination routine sets the record number in the call block for the saved data file. This is to set end-of-file. The routine then closes all files and devices.

13.2.3.26 CVITXT.

This assembly language routine contains all message text. Each message must be in its own CSEG so that it can be accessed by Pascal.

13.2.3.27 CVIUNL.

This routine unloads the source and destination disk volumes. It does not try to unload the system disk. If a bad volume name error is returned, it is ignored.

13.2.3.28 CVIVER.

This routine calls CVIDEV to verify the source, destination, and listing device names.

13.2.3.29 CVIWRT.

This routine copies parameter information from PARM_ARRAY into the output buffer for the saved data file and computes the size of the buffer. It calls CVILUN to assign LUNOs and open necessary devices. The parameter returned from CVILUN contains the LUNOs it assigned. They are also put into the output buffer. The buffer is then written to the saved data file.

13.2.4 CVINIT Debug Suggestions.

SVC Errors
The best place to trap error conditions is in CVIERR. All errors detected by CVINIT pass through CVIERR. These are usually SVC errors. Use the Pascal debugger to find out which module called CVIERR. After assigning a breakpoint at the beginning of CVIERR(ABP), execute the Resume Task (RT) command. When the breakpoint is hit, use the Show Pascal Stack (SPS) command to see the call chain. After finding out which module called CVIERR, you can then run the task again and breakpoint on the SVCs in the calling module.

End Action
If the task takes end action, you need to isolate the module that caused the offense. This is usually a tedious process. One common cause of end action in CVINIT is execution of a privileged instruction when CVINIT has been rendered unprivileged. It should be a privileged task and is shipped in that condition.

The Data Structure Method
If you are not sure about the information that SCI passed to CVINIT, look at the data in PARM_ARRAY (use the SP debug command).

The File Method
If you are not sure about the information that CVINIT passed to CV or BDD, look at the file .S$CV or .S$BDD. CV and BDD look at the data in these files to determine what to do.

When All Else Fails
If, after repeated attempts, you can not find out where the error condition came from (especially if there is no error, but CVINIT is simply doing things wrong), use the Pascal debugger to breakpoint on the beginning and end of each module until you find the offending module. The CV algorithm below can help you trace the flow of control.

3. Initialize all variables.

4. Copy selected fields from the source volume information into the target volume information.

5. Move the track 1 loader, if present.

6. Construct the free ADU list from the bad ADU list from the target disk.

7. Allocate disk space for VCATALOG, and stack VCATALOG.

8. Place the modified volume information on track 1, sector n-1 of the target disk.

9. Put $$$$$$$$ into the volume name and 2 (disk needs INV) into the state flag on track 0, sector 0 of the target disk.

13.3.1.3 Algorithm for the Copy Driver.

1. If an IRB has finished, release it and update the status of the data structure that was using the IRB.

2. Look at all of the common overhead records (CORs) for the FDR and I/O buffers. If its status is:

BUF_NEED_WRIT	- Write the output buffer.
SRC_FDR_FINI	- Write the FDR buffer.
DST_WRIT_FINI	- Read into a verify buffer.
DST_READ_FINI	- Verify the buffer.
BUF_VERF_FINI[IO]	- Set status to BUF_NEED_FILE.
BUF_VERF_FINI[FDR]	- Read in more FDRs.
SRC_READ_FINI[FDR]	- Process the FDRs in the buffer.

3. Look at a buffer allocation record (BAR). If its source read has finished, or if it is in the process of being formatted, try to format the portion of the file it represents.

13.3.1.4 Algorithm for Copying Directories.

1. Stack VCATALOG.

2. If a free FDR buffer and IRB can be found, pop an entry

off the directory stack and read the FDRs it points to from the source disk. If all of the directory could not be read, update the entry and push it back on the stack.

3. If a source read into an FDR buffer has finished, then process the FDRs in the FDR buffer (stack the directory FDRs and process the files).

4. If all FDRs in a buffer have been processed, write the buffer to the target disk.

5. Repeat steps 2 through 4 until there are no more directories.

13.3.1.5 Algorithm for Copying Files.

1. Try to get an additional FDR record (AFR), otherwise leave and try again later.

2. Put the computed target parameters and the disk allocation information into the AFR.

3. Continue initiating reads of the file into the input buffers until no IRBs are left, no BARs are left, no buffer space is left, or no file is left.

4. Format the file into an output buffer.

5. Repeat steps 2 through 4 until all of the file is completely formatted.

6. Using the target parameters, update the FDR in the FDR buffer that represents this file.

13.3.1.6 Algorithm for Copying Program Files.

1. Try to get an AFR, otherwise leave and try again later.

2. Reserve a sector in an input buffer to be treated as a temporary buffer, which will be used to read and write the program file's overhead records.

3. Read and write overhead records until an entry for an image is found.

4. Load another AFR so the image will go through normal file processing as an image file. However, do not allow it to go through EOF processing.

5. Repeat steps 3 through 4 until all of the overhead

records and images have been moved.

6. Using the target parameters, update the FDR in the FDR buffer that represents this file.

13.3.2 CV Data Structures.

The following paragraphs describe the CV data structures.

13.3.2.1 AFR_REC_DEFN.

This holds the additional FDR information. It contains the new target parameters, such as the target physical record length and the target disk allocation. It is also the bookkeeper for all operations performed on the file, such as formatting. An FDR is assigned the same AFR throughout its copy operation.

```
     *---------+---------*
>00  |       AFRSTA      |  Current status of this AFR.
     +---------+---------+
>02  |       AFRTYP      |  File and conversion type.
     +---------+---------+
>04  |       AFRFDP      |  Address of this AFR's FDR.
     +---------+---------+
>06  |       AFRODD      |  Is seq log rec len odd or even?
     +---------+---------+
>08  |       AFRLRI      |  Formatting index into the input buf.
     +---------+---------+
>0A  |       AFRSLL      |  Src logical record length in words.
     +---------+---------+
>0C  |       AFRWSL      |  Words moved from src log rec to dst.
     +---------+---------+
>0E  |       AFRSPL      |  Src physical record length in words.
     +---------+---------+
>10  |       AFRWSP      |  Words moved from src phy rec to dst.
     +---------+---------+
>12  |       AFRSPS      |  Words of slop at end of src phy rec.
     +---------+---------+
>14  |       AFRSAS      |  Words of slop at end of FDRAPB src ADUs.
     +---------+---------+
>16  |       AFRSPM      |  Src phy recs moved, for slop detection.
     +---------+---------+
>18  |       AFRSPU      |  Src phy recs to move before ADU slop.
     +---------+---------+
>1A  |       AFRDLL      |  Dst logical record length in words.
     +---------+---------+
>1C  |       AFRWDL      |  Words moved into a dst log rec.
     +---------+---------+
>1E  |       AFRDPL      |  Dst phy rec length in words.
     +---------+---------+
     +---------+---------+
>20  |       AFRWDP      | Words moved into dst phy rec.
     +---------+---------+
>22  |       AFRDPS      | Words of slop at end of dst phy rec.
     +---------+---------+
>24  |       AFRDAS      | Words of slop at end of dst FDRAPB ADUs.
     +---------+---------+
>26  |       AFRDPM      | Dst phy recs moved, for slop detection.
     +---------+---------+
>28  |       AFRDPU      | Dst phy recs before ADU slop.
     +---------+---------+
>2A  |       AFRPHI      | Index of seq phy rec header.
     +---------+---------+
>2C  |       AFRLHI      | Index of seq log rec header.
     +---------+---------+
>2E  |       AFRLRS      | Status of this dst logical record.
     +---------+---------+
>30  |       AFRLRN      | Will src log rec need combining?
     +---------+---------+
>32  |       AFRLRC      | Combine next src log rec?
     +---------+---------+
>34  |       AFRBSS      | Status of blank suppressed log rec.
     +---------+---------+
>36  |       AFRWBS      | Words moved from blk suppressed log rec.
     +---------+---------+
>38  |       AFRBSL      | Blank suppressed rec length.
     +---------+---------+
>3A  |       AFRPHD      | Dst physical record header word.
     +---------+---------+
>3C  |       AFREOM      | Number of source logical records.
     +---------+---------+  Decrements to zero during formatting.
     /         /         /
     /         /         /
     +---------+---------+
>40  |       AFRBKM      | Dst physical records formatted.
     +---------+---------+
     /         /         /
     /         /         /
     +---------+---------+
>44  |       AFRPAA      | Dst primary allocation ADU.
     +---------+---------+
>46  |       AFRPAS      | Dst primary allocation size in ADUs.
     +---------+---------+
>48  |       AFRAPB      | Dst ADUs per physical record.
     +---------+---------+
>4A  |       AFRBPA      | Dst physical records per ADU.
     +---------+---------+
>4C  |       AFRCRP      | Address of COR of output buffer.
     +---------+---------+
      +---------+---------+
>4E   |       AFRSTI      |  Index into src sec alc table.
      +---------+---------+
>50   |       AFRSOA      |  Source sector offset for next read.
      +---------+---------+
>52   |       AFRSAA      |  Source ADU address for next read.
      +---------+---------+
>54   |       AFRSSA      |  Source sectors left in allocation.
      +---------+---------+
      /         /         /
      /         /         /
      +---------+---------+
>58   |       AFRDTI      |  Index into dst sec alc table.
      +---------+---------+
>5A   |       AFRDOA      |  Dst sector offset for next format/write.
      +---------+---------+
>5C   |       AFRDAA      |  Dst ADU address for next format/write.
      +---------+---------+
>5E   |       AFRDSA      |  Dst sectors left in this allocation.
      +---------+---------+
      /         /         /
      /         /         /
      +---------+---------+
>62   |       AFRDST      |  Dst secondary allocation table.
      +---------+---------+
```

13.3.2.2 AFR_ARRAY.

Holds MAXIIO+1 number of AFRs.

```
   AFR_ARRAY = PACKED ARRAY[1..MAXAFI] OF AFR_REC_DEFN.
```

13.3.2.3 APR_REC_DEFN.

This is used as a variant for an AFR. It stands for Additional Program File Information Record. Its purpose is to hold information about the program file being moved. It is the bookkeeper. It manages the target allocations and the information gathered from the program file record 0. The first three fields match an AFR, the rest have no relation. In the code, an AFR is type changed to be used as an APR.

```
    *---------+---------*
>00 |       APRSTA       | Current status of the program file.
    +---------+---------+
>02 |       APRTYP       | Type of the file(matches AFR).
    +---------+---------+
>04 |       APRFDP       | Address of the FDR attached to this APR.
    +---------+---------+
>06 |       APRAFI       | Index of the AFR used to move images.
    +---------+---------+
>08 |       APRSPA       | Source primary allocation ADU.
    +---------+---------+
>0A |       APRSPS       | Source primary allocation size in ADUs.
    +---------+---------+
>0C |       APRSTI       | Index into the src secondary allocation.
    +---------+---------+
>0E |       APRSOA       | Sector offset for src read of overhead r
    +---------+---------+
>10 |       APRSAA       | ADU for src read of an overhead rec.
    +---------+---------+
>12 |       APRSSA       | Source sectors left in current allocatio
    +---------+---------+
    /         /         /
    +---------+---------+
>16 |       APRDRN       | Number of next free rec in program file.
    +---------+---------+
>18 |       APRDSL       | Sectors left at end of previous image.
    +---------+---------+
>1A |       APRDSA       | ADU address for next write.
    +---------+---------+
>1C |       APRDSO       | Sector offset for next write.
    +---------+---------+
>1E |       APRPAA       | Target primary allocation ADU.
    +---------+---------+
>20 |       APRPAS       | Target primary allocation size in ADUs.
    +---------+---------+
>22 |       APRRNM       | Rec number of overhead record in buffer.
    +---------+---------+
     +---------+---------+
>24  |       APRNDE      |  Maximum number of entries.
     +---------+---------+
>26  |       APRRDE      |  Record number where entries begin.
     +---------+---------+
>28  |       APRODE      |  Offset where entries begin.
     +---------+---------+
>2A  |       APRPRI      |  Index into APRAPR.
     +---------+---------+
>2C  |       APRNAA      |  Total allocated ADUs.
     +---------+---------+
>2E  |       APRDTI      |  Index into target sec alc table.
     +---------+---------+
>30  |       APRDST      |  Target secondary allocation table.
     +---------+---------+
     /         /         /
     /         /         /

     +---------+---------+
>70  |       APRAPR      |  Array of locations of T/P/O/Hole entries.
     +---------+---------+

     +---------+---------+
>70  |       APRMNT      |  Maximum number of task entries.
     +---------+---------+
>72  |       APRTO       |  Offset to beginning of task entries.
     +---------+---------+
>74  |       APRTR       |  Record where task entries begin.
     +---------+---------+
>76  |       APRMNP      |  Maximum number of proc entries.
     +---------+---------+
>78  |       APRPO       |  Offset to beginning of proc entries.
     +---------+---------+
>7A  |       APRPR       |  Record where proc entries begin.
     +---------+---------+
>7C  |       APRMNO      |  Maximum number of overlay entries.
     +---------+---------+
>7E  |       APROO       |  Offset to beginning of overlay entries.
     +---------+---------+
>80  |       APROR       |  Record where overlay entries begin.
     +---------+---------+
>82  |       APRMNH      |  Maximum number of hole entries.
     +---------+---------+
>84  |       APRHO       |  Offset to beginning of hole entries.
     +---------+---------+
>86  |       APRHR       |  Record where hole entries begin.
     +---------+---------+
```

13.3.2.4 BAD_ADU_REC.

This is used to hold an ADU range that has been declared bad by some routine, and which will be used during the pathname trace to find the directories or files that are within that range.

```
    *---------+---------*
>00 |       BGNBAD      | First bad ADU.
    +---------+---------+
>02 |       ENDBAD      | Number of words in bad ADU range.
    +---------+---------+
```

13.3.2.5 BAD_ADU_ARRAY.

This is used to hold a finite number of bad ADU ranges.

```
   BAD_ADU_ARRAY = ARRAY[1..MAXBDI] OF BAD_ADU_REC.
```

13.3.2.6 BAR_REC_DEFN.

This is used to hold information about a file's input buffer allocation. It is used when reading, and when formatting the file.

```
    *---------+---------*
>00 |       BARSTA      | Status of this BAR.
    +---------+---------+
>02 |       BAREOF      | EOF in this buffer allocation.
    +---------+---------+
>04 |       BARAFI      | Index of the AFR currently using this BAR.
    +---------+---------+
>06 |       BARIAB      | Beginning of buffer allocation index.
    +---------+---------+
>08 |       BARIEB      | End of buffer allocation index.
    +---------+---------+
>0A |       BARIRI      | Index of the IRB this BAR is using.
    +---------+---------+
>0C |       BARCRP      | Address of the input buffer's COR.
    +---------+---------+
```

13.3.2.7 BAR_ARRAY.

Is used to hold the BAR_REC_DEFNs. It is a circular list used by the file-reading and file-formatting code. There is one more BAR than there are IRBs (MAXIIO). The array's base is 0, which allows easy index advancement ((cur index + 1) MOD max index).

```
    BAR_ARRAY = ARRAY[0..MAXBAI] OF BAR_REC_DEFN.
```

13.3.2.8 COR_REC_DEFN.

This is the common overhead record structure for all of the I/O, FDR, and VERIFY buffers. It holds the necessary information to enable formatting, I/O, and FDR processing. Its variants enable it to be used for any of the buffer types. Note that since all output and verification processes act on buffers, this common structure allows the use of much common code.

```
     *---------+---------*
>00  |       CORSTA      |  Status of the buffer.
     +---------+---------+
>02  |       CORTYP      |  Type of buffer.
     +---------+---------+
>04  |       CORDOW      |  Dst sector offset for next I/O.
     +---------+---------+
>06  |       CORRII      |  Index of current read IRB.
     +---------+---------+
>08  |       CORWII      |  Index of current write IRB.
     +---------+---------+
>0A  |       CORBPT      |  Address of the buffer.
     +---------+---------+
>0C  |       CORVCP      |  Address of the verify buffer COR.
     +---------+---------+
>0E  |       CORAWB      |  Words currently used.
     +---------+---------+
>10  |       CORDAW      |  Dst ADU address for the next I/O.
     +---------+---------+
>12  |       COREC       |  Number of times this error has occured.
     +---------+---------+


     +---------+---------+
>14  |       CORRFB      |  Number of FDRs in this buffer.
     +---------+---------+
>16  |       CORFDI      |  Index of first file FDR.
     +---------+---------+
>18  |       CORLFI      |  Index of last file FDR.
     +---------+---------+


     +---------+---------+
>14  |       CORTWB      |  Length of buffer in words.
     +---------+---------+
>16  |       CORCRP      |  Address of input buffer COR.
     +---------+---------+
>18  |       CORIEB      |  Index following last allocated word.
     +---------+---------+


     +---------+---------+
>14  |       CORPFS      |  Program file buffer status.
     +---------+---------+
>16  |       CORAFI      |  Index of APR (AFR).
     +---------+---------+
```

13.3.2.9 DER_REC_DEFN.

This is a directory stack entry. It holds the number of FDRs left in the directory, and the source and target disk addresses to read from and write to.

```
    *---------+---------*
>00 | DERSOR  | DERDOW  | Sector offset for the next source read.
    +---------+---------+   Sector offset for the next target write.
>02 |       DERRFB      | Number of remaining FDRs in the directory.
    +---------+---------+
>04 |       DERSAR      | ADU address for the next source read.
    +---------+---------+
>06 |       DERDAW      | ADU address for the next target write.
    +---------+---------+
```

13.3.2.10 DER_ARRAY.

This is used as a stack of DER_REC_DEFN. It allows the program to access directories at random. So, directories are not read sequentially, but may be partially read and written. If CV aborts, directories may be left partially filled on the target disk.

```
    DER_ARRAY = ARRAY[1..MAXDRI] OF DER_REC_DEFN.
```

13.3.2.11 FAR_REC_DEFN.

This is used to hold a free ADU range that is on the target disk.

```
    *---------+---------*
>00 |       FARNFA      | Number of free ADUs in this range.
    +---------+---------+
>02 |       FARAFA      | ADU address of free ADU range.
    +---------+---------+
```

13.3.2.12 FAR_ARRAY.

This is used to hold a finite number of free target ADU ranges. Currently, it will hold up to 66 ranges.

```
    FAR_ARRAY = ARRAY[0..MAXFRI] OF FAR_REC_DEFN.
```

13.3.2.13 IRR_REC_DEFN.

This is used to hold a user call block for disk I/O and to hold the status and error count for that call block.

```
     *---------+---------*
>00 |       IRRSTA       | Status of the call block.
     +---------+---------+
>02 |       IRREC        | Number of times this I/O has failed.
     +---------+---------+
>04 |       IRRIRB       | User call block.
     +---------+---------+
     /         /         /
     /         /         /
```

13.3.2.14 IRR_ARRAY.

This is used to hold a finite number of IRR_REC_DEFN for both the source and target disk I/O.

```
    IRR_ARRAY = PACKED ARRAY[1..MAXIIO*2] OF IRR_REC_DEFN.
```

13.3.2.15 LEVEL_ENTRY.

This is used during the pathname trace to keep track of the directories being read.

```
     *---------+---------*
>00 |      LEVDNM        | Name of current directory on this level.
     +---------+---------+
>08 |      LEVRFD        | Number of FDRs still to be read.
     +---------+---------+
>0A |      LEVRAD        | ADU address of remaining FDRs.
     +---------+---------+
>0C |      LEVROF        | Sector offset of remaining FDRs.
     +---------+---------+
```

13.3.2.16 LEVEL_ARRAY.

This is used during the pathname trace to retain information about the directories being read at various levels.

```
   LEVEL_ARRAY = ARRAY[0..23] OF LEVEL_ENTRY.
```

13.3.2.17 PFIARR.

This represents the maximum number of entries, which describe images, that an overhead record in a program file can hold. Currently this is 16 16-byte entries.

```
   PFIARR = PACKED ARRAY[0..MAXPFI] OF PFI.
```

13.3.2.18 PRA_REC_DEFN.

This is a definition of an entry in the first overhead record of a program file. It is used to tell where the entries for tasks, procedures, overlays, and free spaces begin, and the maximum number of entries each can hold.

```
    *---------+---------*
>00 |       PRAMNE      | Maximum number of entries.
    +---------+---------+
>02 |       PRARCO      | Record where entries begin.
    +---------+---------+
>04 |       PRARCA      | Offset in record where entries begin.
    +---------+---------+
```

13.3.2.19 PRAARR.

This holds the PRA_REC_DEFNs from the overhead record 0 of a program file.

```
   PRAARR = PACKED ARRAY[1..MAXPRI] OF PRA_REC_DEFN.
```

13.3.2.20 PRR_REC_DEFN.

This is used to hold a record from the parameter file created by CVINIT (the initial task that interacts with SCI). These records are allocated space and are read in the initial routine of CV.

```
    *---------+---------*
>00 |       PRRLAR      |  LUNOs for this copy
    +---------+---------+
    /         /         /
    +---------+---------+
>04 |       PRRSDN      |  Source device name.
    +---------+---------+
    /         /         /
    +---------+---------+
>08 |       PRRDDN      |  Target device name.
    +---------+---------+
    /         /         /
    +---------+---------+
>0C |       PRRLDN      |  Listing device name.
    +---------+---------+
    /         /         /
    +---------+---------+
>10 |       PRRSVN      |  Source volume name.
    +---------+---------+
    /         /         /
    +---------+---------+
>18 |       PRRDVN      |  Target volume name.
    +---------+---------+
    /         /         /
    +---------+---------+
>20 | PRRVER  | PRRMCP  |  Verify copy?
    +---------+---------+     More copies?
>22 | PRRCRR  | PRRCSF  |  Convert rel recs/prog files?
    +---------+---------+     Convert seq files?
>24 | PRRRRN  | PRRFL1  |  Run number in ASCII.
    +---------+---------+     Fill byte.
```

13.3.2.21 PRR_ARRAY.

This is used to hold up to nine records from the parameter file created by CVINIT (the initial task that interacts with SCI). See PRR_REC_DEFN.

```
PRR_ARRAY = ARRAY[1..MAXRRN] OF PRR_REC_DEFN.
```

13.3.2.22 SAT_REC.

This is a definition of a secondary allocation entry in an FDR.

```
    *---------+---------*
>00 |       SATSAS      |  Number of ADUs in the allocation.
    +---------+---------+
>02 |       SATSAA      |  ADU address of the allocation.
    +---------+---------+
```

13.3.2.23 SAT_TBL.

This is used to hold a finite number of secondary allocation records. Currently the maximum is set at 16 in an FDR.

```
   SAT_TBL = PACKED ARRAY[1..MAXSTI] OF SAT_REC.
```

13.3.3 CV Pascal Modules.

The following paragraphs briefly discuss each of the CV modules written in Pascal.

13.3.3.1 CV.

This is the main routine that calls the assembly language routines to get memory, allocate the memory to the various buffers, and to read in the parameter file. It calls the routines that initialize, perform, and complete a copy operation, looping until all copies are done or until the user wants to quit.

13.3.3.2 CVALCA.

This routine allocates ADUs on the target disk, using the free ADU list that was created using the bad ADU list from the target disk.

13.3.3.3 CVALCD.

This routine allocates disk space for all requested ADUs, and returns the disk allocation in two primary allocation variables and/or a secondary allocation table.

13.3.3.4 CVBIAS.

This routine converts binary numbers to ASCII decimal or hexadecimal characters.

13.3.3.5 CVCDEV.

This routine prompts the user to mount the volume if one of the following situations occurs:

* This is not the first copy.

* It is the first copy and the system-disk-flag is set.

* The volume specified by the user is not in the drive.

* An error occurs reading the drive.

If everything is correct, it reads track 0 sector 0, VCATALOG's DOR, and the disk information into the first three sectors of the specified buffer.

13.3.3.6 CVCDIO.

This routine issues all I/O on the target device for the FDR and I/O buffers. It updates the status to indicate whether it began a write from a buffer or read into a verify buffer.

13.3.3.7 CVCFLE.

This routine processes file I/O and formatting errors.

13.3.3.8 CVCPRM.

This routine computes all of the target parameters necessary for updating the FDR, formatting the file, and allocating target disk space. It must also make sure everything is properly initialized for the above three functions. All final values are in words, not bytes. Some primary computations are:

Destination Physical Record Length	Either the default physical record length For the target disk, or an ADU, depending upon which packs logical records best. For sequential file, if the target physical record length is greater than the output buffer length, the target physical record length is made equal to the target sector length. This ensures proper formatting.
Slop Values	The amount of unused space at the end of source or target physical records and the unused space caused by packing physical records into ADUs.
Formatting Unit	The length of the indivisible unit to be used when formatting. When no

	conversion of physical record length is being performed, this is the physical record length. Otherwise, it is the logical record length. This unit is of variable length when converting sequential files.
Destination File Length	The length of the formatted file in target ADUs.

In certain cases, the actual physical record length, the amount of data in the physical record, and the physical record length rounded to a multiple of sectors must be known. The only unusual aspect of this routine is the passing of data for relative record file formatting in two fields of an AFR (AFRBSL and AFRPHD) that are only to be used during the formatting of sequential files.

13.3.3.9 CVCSCY.

This routine computes the number of sectors that are left on a cylinder.

13.3.3.10 CVCSNW.

This routine computes the number of words that CVFFOR should move for a sequential logical record. At the beginning of a source physical record, CVCSNW checks to see if the last logical record is split. At the beginning of a target physical record, it remembers the index so it can update the header word at a later time. It advances the output buffer index. In order to process a logical record, CVCSNW must determine if the record must be split across target physical records, or if it must combine two source logical records. Note that it may be splitting a record it is combining or combining a split record. If the logical record is blank suppressed, the record is moved one blank-suppressed record at a time. Blank-suppressed records can be split across target physical records.

13.3.3.11 CVCSRD.

This routine pops a directory entry off the directory stack and initiates a read into an FDR buffer.

13.3.3.12 CVCSVC.

This routine processes I/O SVC errors. If the error has not happened MAXERR times, it will reissue it. If it has happened MAXERR times, an error message is printed and the ADUs represented by the IRB are put into the bad ADU list.

13.3.3.13 CVCVER.

This routine processes verification errors. It allows the buffer to be written/read/verified MAXERR times before it declares the verification bad, prints an error message, and puts the bad ADU range in the target bad ADU list.

13.3.3.14 CVCVOL.

This copy driver routine is basically a busy wait. It checks all of the SVC call blocks to see if any have finished. It then updates the appropriate records if any I/O has completed. This routine calls the routines that format and allocate target disk space for files, read/write/verify FDR and I/O buffers, and initiate all source reads of files. In order to avoid using the Pascal linkage routines, many of the functions that would normally be delegated to other modules have been incorporated into this routine.

13.3.3.15 CVENDR.

This routine builds and prints the end-of-run message. It prompts the user to ask if they wish to continue to the next copy. If necessary, it calls the bad file pathname trace.

13.3.3.16 CVFFOR.

This routine formats all file types. Note that program files will come through here one image at a time. Sequential files also go through the routine CVCSNW to determine how many words to move and to set flags for any additional processing required. CVFFOR will try to format only the portion of the file that is represented by the current-format input buffer allocation record. This forces the files to be formatted in the same order that they were read. As the portion of the file that the BAR represents is formatted, the BAR is released, enabling an AFR to use the BAR to initiate another file read at a later time.

If a file is being converted, it is moved by logical records. If it is not being converted, it is moved by physical records. The formatting routine treats the conversion of sequential files as a special case, since they have variable length logical records. All other file types are treated normally.

Beginning at the record index into the input buffer (AFRLRI), CVFFOR tries to move a logical record. If it has moved all of the data in a source physical record, it advances the source buffer index past any slop that might be at the end of the physical record. If it has moved all of the source physical records that have been packed into an ADU(s), it advances the index past any slop there. The same goes for the index into the output buffer when a target physical record has been filled. If either of the indexes is advanced past the allocation in its buffer, either by skipping slop or moving data, the formatting stops and enough information is retained so that after getting more source or freeing an output buffer, formatting can continue upon reentry.

Since a sequential logical record can be split across physical records, and perhaps combined while changing record lengths, sequential data movement is treated as a special case at points where the header and trailer words can be updated appropriately.

13.3.3.17 CVFNC1.

This routine performs the calls to the Pascal divide/multiply routines. Specifically, this routine makes calls to divide words, divide words rounded, and round OP1 up or down to a multiple of OP2.

13.3.3.18 CVFNC2.

This routine adds/subtracts OP1 from OP2 and then DIVs/MODs by OP3.

13.3.3.19 CVFNC3.

This routine performs long integer division rounded up.

13.3.3.20 CVFSRD.

This routine issues as many reads of a file into the input buffers as possible. It will stop trying when any of the following occur:

* There is no buffer space.
* There are no IRBs.
* There are no BARs.
* All of the file has been read.

13.3.3.21 CVGIOB.

This routine gets an output buffer for formatting.

13.3.3.22 CVLAFR.

This routine loads an AFR with information about an image on a program file from the additional program file information record (APR) and the FDR of the program file. This AFR goes through normal file processing except that it does not go through end-of-file processing. Only one image of a program file is moved at a time.

13.3.3.23 CVPDIR.

This routine formates an FDR buffer, stacks and allocates disk space for directories, and sets the file FDR indexes.

13.3.3.24 CVPERM.

This routine builds and prints error messages.

13.3.3.25 CVPHDR.

This routine prints the various headers on the interactive and listing devices.

13.3.3.26 CVPPGF.

This routine processes the overhead records of a program file. It also sets up the AFR for moving the program file image. Note that little parallel I/O occurs during program file movement.

13.3.3.27 CVPSTA.

This routine calls the routines that print the status line on the VDT screen.

13.3.3.28 CVPSVC.

This routine issues all I/O with an SVC call.

13.3.3.29 CVRWVR.

This routine performs either a read SVC or a write/read/verify SVC sequence. It allows MAXERR retries on I/Os and verifications.

13.3.3.30 CVSMAP.

This routine updates the target disk partial bit maps and puts the disk into a usable state. It first sets all of the ADUs in a map, as allocated, and then resets the ADUs that the free ADU list (FARARP) says are not allocated.

13.3.3.31 CVSTR1.

This is the outer level initialization routine for each copy. It is called at the beginning of each copy. It reads the target's bad ADU list, sets up the free ADU list, opens the devices, allocates disk space for VCATALOG, stacks VCATALOG, and copies the track 1 loader. It calls the routines that check the volumes in the drives and initialize the data structures.

13.3.3.32 CVSTR2.

This copy initialization routine loads the disk parameter variables which are in common with the information from buffers loaded in CVCDEV. It also initializes IRBs, AFRs, BARs, CORs, and the stack (DERARP).

13.3.3.33 CVTRCP.

This pathname trace routine determines which files on a given volume have bad ADUs in them.

13.3.4 CV Assembly Modules.

The following paragraphs briefly describe the CV assembly language modules.

13.3.4.1 CVCMPB.

This routine compares two buffers, given the number of words to verify.

13.3.4.2 CVDMVB.

This routine moves words from one buffer to another. Words are moved from the bottom up.

13.3.4.3 CVEACT.

This routine executes if the task takes end action. It formats a message and calls the end-of-copies routine to terminate the task cleanly. This routine traps the Pascal label R$EACT, so that the Pascal compiler will put this routine's R$EACT label into word 3 of the task image.

13.3.4.4 CVENDC.

This is the termination (end-of-copies) routine. It prints an error message if one is passed to it. It then issues the end-of-copies message and, if the system disk was involved in any copy, hangs the system. If the system disk was not involved, it issues an end of task SVC.

13.3.4.5 CVLCOM.

This module only lists some of the variables in common because CVSTRT (the initialization module) cannot assemble with all of them in it. The discussion of CVSTRT explains why all common variables must be referenced before the initialization code. reserve block5

13.3.4.6 CVMOVB.

This routine moves words from one buffer to another, given the offsets from which to move and the number of words to move.

13.3.4.7 CVMSGM.

This routine contains all the text messages used by CV.

13.3.4.8 CVPMSG.

This routine issues all I/O to the interactive and listing devices.

13.3.4.9 CVPTCH.

This is the patch module.

13.3.4.10 CVRRTE.

This routine traps Pascal abort and exit routine labels and SCI calls. It formats a message and calls the end-of-copies routine.

13.3.4.11 CVSTRT.

This module gets memory for buffers and reads the parameter file. This code is used as buffer space and is overwritten during execution. In order to do this, the code had to be at the end of the task. This was made possible by putting the code into a CSEG, and by making sure all other common variables were referenced before the code in the code-CSEG. This insures that the code will be the last thing in the task space. Note also that the Pascal stack-getting routine has been replaced here, and the Pascal stack is a static structure in this code.

13.3.5 CV Special Cases.

Certain file types, individual files, and directories are treated as special cases by CV. The following paragraphs discuss these special cases.

13.3.5.1 Special Case File Types.

Program files, image files, and key indexed files have various exceptions to standard CV operations. The exceptions for each file type are as follows:

1. Program Files - Each image of a program file is allocated contiguously and each image goes through normal file processing with an AFR that makes it look like an image file. It does not go through end-of-file processing. Instead, control goes back to CVPPGF so that the rest of the program file can be moved.

2. Image Files - Image files are allocated contiguous space on the target disk. Their physical record length is set to a target sector length.

3. KIF Files - KIF files are never compressed, and their physical records are never converted. The copy is done by physical record, and the target disk allocation is made as if the file were bounded.

13.3.5.2 Special Case Files and Directories.

CV treats the following files as special cases:

1. .S$OVLYA - Compress, but never convert its physical record length.

2. .S$TCALIB - Compress, but never convert its physical record length.

3. .S$CLF - Compress, but never convert its physical record length.

4. .GENDAT - Compress, but never convert its physical record length.

5. .JENDAT - Compress, but never convert its physical record length.

6. .S$ROLLA - Never compress, but physical record length can be converted.

7. .S$DIAG - Allocate space for this file on the last available ADUs on the target disk.

8. .VCATALOG directory - In CVPDIR, update .VCATALOG's FDR, but do not stack it.

13.3.6 CV Debug Suggestions.

To change the size of the directory stack, patch the value of the common variable MAXDRI in CVSTRT.

To change the size of the stack, R$GSHP and R$GSHS (in CVSTRT) must return the new value, and references to CSGINT must be altered to correspond to the new stack length. CSGINT is defined to be the part of the code that can be used as buffer space and which begins at the end of the stack. This patch is not easy, but only requires changing four words (in line) in CVSTRT.

To change the size of the FDR buffers, patch the value of the common variable MAXFDR in CVSTRT.

The best places to breakpoint an error condition caught by CV are in CVCSVC, CVCFLE, or CVCVER. These are the only modules that process errors during a copy.

If the error is a file formatting error (CVCFLE called from CVFFOR), the AFR assigned to that file contains all of the pertinent file information. To find the correct AFR, use the index indicated in the BAR passed to CVCFLE. The AFR is your best bet for file allocation or formatting problems.

If CV does not catch the error, and it is a file formatting error, the best thing to do is try a copy of that one file. Breakpoint the code before and after CVFFOR. This allows you to examine the formatting process as it proceeds. The BAR indexed by CFRBRI is the current BAR that needs formatting. It contains the index for the AFR.

If the error occurs when printing messages, the problem may be in CVPMSG. This is the only message printer. However, CVPERM and CVPSTA are responsible for building a message with variable text and numbers to be passed to CVPMSG.

The majority of errors involve file formatting. If it appears that not enough disk space was reserved, then CVCPRM is responsible. However, if it appears that allocation of disk space was the problem, see CVALCD and CVLACA.

Be very careful altering the sequential file formatting code. Much of it is context sensitive. The best safeguard is to work through, by hand, an example in which a source blank-suppressed logical record, with an odd length, is split across source physical records, but is combined and split across target physical records. Such an example will illustrate the reason for the placement and setting of the various status flags.

Note that any additions to the modules CVFFOR or CVCSNW will cause a significant reduction in the speed of CV. Module calls within the formatting loop in CVFFOR should be avoided. R$LINK adds a major detour to the code path.

13.4 BACKUP DIRECTORY TO DEVICE (BDD)

The following paragraphs contain the design specification for the BDD utility. This utility allows users to perform a rapid sequential copy of the data in a disk directory or volume. The destination device may be either disk or tape and the backup file format is compatible with the Restore Directory (RD) processor.

BDD is bid by CVINIT, the initial task for BDD and CV, and runs in foreground mode. BDD performs a series of up to nine backup operations from a disk volume or directory to a disk or tape. The user is kept aware of BDD's progress by a VDT display which keeps a running total of the number of bytes and files transferred to the destination media. The numbers displayed on the screen represent the actual number of bytes of data written. This number may be different than the number of bytes of actual source, since BDD performs data compression (when possible) for source files with disk space allocated to them beyond the last physical record in the file.

13.4.1 BDD Data Structures.

All of BDD's data structures are declared as COMMON variables. BDD is compiled with the LOCALS Pascal compiler option to achieve better (smaller) task size. Therefore much communication between BDD modules is accomplished through these common variables.

13.4.1.1 Buffers.

BDD uses a total of six large data structures. Their usage is as follows:

* Two READ buffers

* Two I/O buffers

* One VERIFY buffer

* One FDR buffer

The READ, I/O, and VERIFY buffers are identical in structure and their sizes are determined dynamically for each rerun of BDD. These buffers are simply arrays of WORDs which are indexed from 0 to the dynamically determined maximum.

The READ buffers are used to read data from the user's source files. Note that by data we mean actual file data, since FDRs for the files are read into the FDR buffer. The reads of this file data are accomplished by using direct disk I/O (read by ADU) in initiate mode. Therefore the data in these buffers at any point during BDD execution is an in-memory copy of the disk image.

The I/O buffers are used to hold data that has been formatted and moved from a READ or FDR buffer. This data is written directly to the output device. If the output device is a disk, the data is written to a VCATALOG image file using direct disk I/O (write by ADU). If the destination is a tape drive (destination pathname = MTxx), the buffer is written using device I/O. In either case, the write is performed in initiate mode.

The VERIFY buffer is used to hold data read from the destination (after it is written from an I/O buffer) to be compared to an I/O buffer already prepared and in memory. It is important to note that verification compares the formatted in-memory data to the data on the destination. It does not compare the source to the destination. If the destination is a disk, BDD starts (or queues) a read into the VERIFY buffer immediately after writing from an I/O buffer. If the destination is tape, verification is performed as a second pass over the source. That is, BDD does all the work a second time, but at the point where it would have written an I/O buffer in the first pass, it instead reads from the destination into the VERIFY buffer. This second pass is to avoid having the tape flutter back and forth during the backup.

Several common variables are used to support the READ, I/O, and VERIFY buffer. These are IOPTR, IOINDX, RPTR, RINDX, IOVSIZE, and RSIZE. Each of these common variables is an array indexed from 0 to 1 with the exception of IOVSIZE and RSIZE, which are merely positive integers representing the size in WORDs of the I/O and VERIFY buffer and the READ buffers. The size of the I/O and VERIFY buffers must be the same since the VERIFY buffer will be used in a comparison with one of the I/O buffers.

The other array data structures run from 0 to 1 and are indexed by the common CIO (current I/O buffer). The current I/O buffer is the one that will receive data next. IOINDX is the word index into the I/O buffer of the next available word. IOPTR is an array of two pointers to the two I/O buffers. RINDX and RPTR serve a similar function for the two READ buffers.

Note that the size of the I/O and VERIFY buffers must be an even multiple of the sector size of the destination disk. The size of the read buffers must be an even multiple of the sector size of the source disk. If the destination is tape, then BDD creates the I/O buffers the largest size possible that is a multiple of 288.

The FDR buffer is more complicated in its structure than the other large buffers. The FDR buffer is conceptually a stack. Each stack entry is a queue. Each queue entry holds a sector from a source directory file. Therefore the FDR buffer is a stack of queues. The stack size is fixed at 23. Each entry in the stack represents one nesting level in the source pathname currently being backed up. Since the maximum pathname length is 48 characters, the maximum nesting level for any file is 24 (for example, A.B.C.D.E.F... and so on). The VCATALOG level is not contained in the FDR buffer, thus the 23 stack entries. Currently each level queue holds two entries.

A diagram of the FDR buffer structure is shown in Figure 13-1.
begin figure

```
                          stack bottom
                     +---------+
                     |   |     |    QUEUE FOR LEVEL 1
                     |---------|
                     |   |     |    QUEUE FOR LEVEL 2
                     |   |     |
         |               .
         |               .            .   FIFO---->
   LIFO  |               .            .  (head of queues)
        \|/          |---------|
                     |   |     |    QUEUE FOR LEVEL N-2
                     |   |     |
  LEVEL STACK        |---------|
                     |   |     |    QUEUE FOR LEVEL N-1
                     |---------|
                     |   |     |
 (top of stack)      |   |     |    QUEUE FOR LEVEL N
                     +---------+
```

Figure 13-1 Interfaces Between SCI and Control Tasks

To produce a backup file compatible with the RD processor, each directory on the source disk must be completely moved to the backup file before any of the data from brother directories is moved. So, if BDD is currently working on a directory at level N, the FDR buffer slots for levels N+1 through 23 are available for use. To take advantage of this fact and avoid extra source reads, BDD reads directories at level N into all the slots from N through 23.

When BDD encounters an FDR for a directory file it is forced to adjust the FDR buffer by moving two (= max sectors per level) FDRs from the deepest level up to their proper level queue, and begin a read of the directory into the next deepest level. This adjustment causes a reread of any source sectors that were thrown away during the adjustment. This reread is forced on BDD by the structure of the backup file required by the RD processor.

The FDR buffer is statically declared in the Pascal source, but its size can be changed by changing the values of two constants, MAX_NEST(= 23), and SECT_PER_LEVEL(= 2). A Pascal variant record structure allows BDD to index into the FDR buffer in one of two ways. The first access method treats the FDR buffer as a two-dimensional array [1..23,1..2]. The second method treats the buffer as a monotone array [1..23*2]. The monotone access is used for FDRs (source sectors) in the FDR buffer that reside on a level lower than their proper level.

At any point during BDD execution, the current FDR (the one then being examined or backed up) is defined by FDRBUF[TOP_Q,Q_HEADS[TOP_Q]]. TOP_Q is an integer indicating which level is current, and Q_HEADS is a 23 entry array of integers indicating the offset from the first entry on level TOP_Q. One other BDD data structure, CURDIR, is logically related to the FDR buffer. CURDIR is an array of records that is indexed from 1 to MAX_NEST(= 23). Each entry in CURDIR tracks the current state of the FDR at offset Q_HEADS[level number] in the FDR buffer. One entry in CURDIR looks like the following:

>00	START	Start ADU for the directory file, this level
>02	LENGTH	Total length of this directory (in sectors)
>06	INBUFF	Total number of sectors now in buffer
>08	TOPADU	ADU address of the first sector, this level
>0A	TOPSEC	Sector offset (from TOPADU), first sector
>0C	EODIN	Boolean indicating logical end of directory
>0E	PNAME	8-character name component of this directory

The START, LENGTH, & PNAME fields remain constant whereas the other fields change when BDD reads the next portion of a directory into the FDR buffer. <TOPADU,TOPSEC> is the disk address of the first sector for the given directory currently in the buffer. EODIN is a boolean that is set when the all of the given directory has passed through the FDR buffer. Since a read of a directory file into the FDR buffer may use up portions of the FDR buffer that properly belong to "lower" levels, the data structure CURDIR is needed to track this overflow into the lower levels.

13.4.1.2 Other BDD Data Structures.

This paragraph briefly explains the uses of some of the other data structures used by BDD, which may or may not be clear from a perusal of the source code.

The following common variables are used when the destination device is a disk rather than a tape drive: BKUPADU, DKUPSEC and DESTFDR.

<BKUPADU,BKUPSEC> is the destination disk address of the FDR for the backup file in the destination's VCATALOG. DESTFDR is a copy of the backup file FDR which is kept in memory until the backup completes normally. A rudimentary FDR is placed on the destination disk immediately after BDD discovers that there is an available slot in the destination VCATALOG. This rudimentary FDR has all the constant fields filled in but omits the allocation information in the FDR. This information is filled in only after the backup completes normally. Therefore, if the backup aborts before normal completion, no space on the destination disk has been allocated to the backup file. Note that BDD has written into free ADUs on the disk, but these areas have not been actually allocated to the backup file until the end of the backup. DESTFDR, the in-memory copy of the FDR, has the allocation information filled in immediately after the rudimentary copy is written to disk; hence DESTFDR's allocation fields are used to calculate the next location to write on the disk.

The U_CHAIN and U_ANCHOR variables are used in moving (and formatting) data from a read buffer to an I/O buffer.

UCHAIN is a chain of records describing unformatted source currently residing in a read buffer. U_ANCHOR is an index which indicates which of the statically allocated links in the chain is first on the list. The format of a link is as follows:

```
U_LINK = record
  RB          : 0..1;          "WHICH READ BUFFER?
  RB_SEC_INDX : SMALLINT;      "SECTOR OFFSET INTO THE READ BUFFER
  LENGTH      : SMALLINT;      "LENGTH OF THIS READ(IN SOURCE SECTORS)
  ADUNUM      : WORD;          "ADU NUMBER OF THE READ
  SECNUM      : POSINT;        "SECTOR OFFSET OF THE READ
  READ_DONE   : boolean;       "HAS READ COMPLETED?
  NEXT   : 0..MAX_INITIATES;   "NEXT LINK IN U_CHAIN(0=NONE)
end;(*U_LINK*)
```

Note that <ADUNUM,SECNUM> defines the source disk address where the portion of the read buffer described by this link was obtained. The READ_DONE field is needed because reads into a read buffer are initiated. Finally, note that the LENGTH and RB_SEC_INDX fields change as the data is moved from a read buffer into one of the I/O buffers.

The following common data structures are used in tracing source pathnames on their way to the backup file: DIREC, DIRECTRIES, LEAFS and LEAVES. This is done so that a report of the pathnames in error can be given to the user in the event of an I/O error on the destination device.

DIREC is a pointer at the array DIRECTRIES, and likewise, LEAFS is a pointer at the array LEAVES. The array LEAVES is treated in the code as if it were declared as follows:

```
  LTRACE = array[0..1,1..MAX_LEAF_TRC]of LTRC_REC;
```

where LTRC_REC is:

```
  LTRC_REC = record
   NAM   : NAME;          "leaf name
   DPART : integer;       "points at entry in DIREC@
  end; (*LTRC_REC*)
```

LEAVES holds the leaf components of the files which are currently in one of the I/O buffers. The eight-character leaf component is stored in the NAM field. The DPART is an index into the DIRECTRIES array. This index specifies where the front part of the pathname (the part which goes in front of the leaf component) may be found. The LEAVES data structure is declared large enough to hold all the leaf components which might possibly be contained simultaneously in both I/O buffers. On the other hand, the DIRECTRIES array is declared only large enough to hold a heuristically determined maximum number of "front ends". This is done to save task code space.

Should the DIRECTRIES array overflow during execution, no message is issued to the user until an I/O error occurs on the destination disk. At that time, if the DIRECTRIES data structure has overflowed, only the leaf components of the pathnames in error will be displayed for the user. If the DIRECTRIES array has not overflowed, BDD will be able to report fully expanded pathnames of all files in the I/O buffer that encountered the error on the destination. Finally, note that the overflow of this array is not important, unless an error occurs getting the data to the destination media. Further explanation of these two data structures may be found in the Pascal type declaration module for BDD, located at DSC.TEMPLATE.PTABLE.BDTYPE.

13.4.2 BDD Program Flow.

1. GET MEMORY FOR BUFFERS AND PARAMETER FILE AND READ PARAMETER FILE

   Errors during this process will cause an abort of the entire program since all reruns depend on the parameter file and memory for the large buffers. Requests for memory for the I/O, READ, and VERIFY buffers are reduced until the desired amount is obtained, or until further reducing the request size would mean fewer than

2304 bytes per I/O buffer. This memory request limit means that BDD can run in approximately 53 K bytes of task space if necessary. When setting up pointers to the free space for large buffers, BDD takes advantage of the fact that the initial request for memory for the parameter file will return a pointer that is essentially the beet size of the task plus one beet. Note that the entire parameter file is read into memory and held until completion of all reruns. (This means that doing multiple reruns per task execution is slower, since memory needed to hold the parameters is stolen from I/O buffers). Also note that one of the bid parameters used by the initial task for bidding BDD is the number of beets required to hold the parameter file. After completion of this portion of BDD, the system disk is no longer needed.

2. SET THE BID PARAMETERS FOR THIS RERUN.

   BDD interprets the in-memory parameter file to set up the following common flags for the rerun:

LUNOS	LUNOs for interactive, listing, source, and destination devices.
PATHS	Pathnames for source and destination.
USESYS	A flag that indicates if the system disk was involved.
VERIFY	A flag that indicates if verification was requested.
DEVICES	An array of device names of the devices to be used.

   It also initializes the LUNO fields of the IRBs dedicated to the source and destination devices with the LUNOs for these devices.

3. OPEN THE LISTING AND INTERACTIVE DEVICES.

   An error on these opens will cause a task abort. The listing and interactive devices are opened with an open rewind operation in order to form feed the listing device and clear the screens of the interactive devices before each rerun.

4. GET THE SOURCE AND DESTINATION DEVICES MOUNTED AND OPENED.

   For the first rerun only, BDD will not issue mount requests for the user if the volumes specified by the parameter are already in the proper drives. For all reruns, the routine BDMONT verifies that the proper volumes have been mounted. Verification of mount completion consists of opening each drive involved, reading the volume information for the disk drives involved, and setting the dynamic buffer sizes for the rerun. For disks, validation consists of checking that the volume name in ADU 0 of the disk matches the volume name specified in the parameter file. For tapes, validation varies depending on whether the mount routine is entered during first pass or second pass processing. During first pass, tapes are validated by issuing a dummy write of the tape, followed by a backspace LUNO. For second pass processing, tape validation consists of a forward space/backward space LUNO sequence. The dummy write during first pass is so that "virgin" tapes will not produce >43 errors on a forward space operation.

5. BUILD FDR FOR BACKUP FILE AND PLACE SKELETAL VERSION ON DISK.

   A skeletal FDR for the backup file is built and placed on the disk. The constant fields of the FDR are filled in, but the information regarding disk allocation for the file is omitted from this skeletal version. After the partial FDR has been placed on the disk, BDD scans the partial bit maps on the destination disk to find the 17 largest free areas. These 17 slots are then placed in the in-memory copy of the FDR which was placed on the disk. Note that the free areas located during this scan must be at least as large as one I/O buffer to qualify for inclusion in the in-memory FDR.

6. FIND THE TOP LEVEL SOURCE DIRECTORY.

   The final start-up operation is to locate the directory to back up on the source disk. BDD begins at VCATALOG on the source disk and searches for the directory (or file) specified by the user. When found, the FDR for the top level directory is left in the FDR buffer. Information about this top level pathname, including start ADU, length in sectors, and the eight character component name is left in the common CURDIR.

7. INITIALIZE THE BUFFER INFORMATION FOR THE LARGE BUFFERS.

   The level queues (in the FDR buffer) are all marked as empty, except for the top level. The actual backup process begins.

8. READ AS MUCH AS POSSIBLE OF THE FIRST DIRECTORY INTO THE FDR BUFFER

   Reads into the FDR buffer are performed as explained in the discussion of BDD data structures. When a directory is read, the amount to read is determined by the space remaining in the FDR buffer from the current level through level 23 (the last level).

9. PROCESS THE ENTRIES IN THE FDR BUFFER.

   Entries in the FDR buffer at the current level are processed in the order in which they are read. Execution continues as long as FDRs for other directories are not encountered. When directory FDRs are found, the FDR buffer is adjusted and a read of the new directory is performed into the the next deeper level in the buffer. When file FDRs are encountered in the FDR buffer, some of the information which is needed to continue the backup process is saved before the file FDR is moved to an I/O buffer for output. Eventually the routine BDMVPR is called to move all of the physical records in the file to an I/O buffer.

10. MOVE THE PHYSICAL RECORDS IN THE FILE THRU AN I/O BUFFER.

    The routine BDMVPR essentially controls all of the actual backup work. Data is read into read buffers, unbuffered (formatted) into an I/O buffer, and written to the destination. When all of the physical records have been moved, control returns to step 9. Note that this routine is speed critical. Each Pascal procedure call added to this routine will cause an approximate 2% overall increase in the time to perform a backup. Care has been taken in the design of this module to execute calls only when absolutely necessary. (This includes calls by Pascal to run-time routines.)

11. CLEAN UP AFTER THE BACKUP.

    When the FDR buffer has been exhausted, the rerun is essentially complete. For disk destinations, cleanup operations consist of the following:

    a. Change the in-memory copy to reflect the actual amount of disk space used by the backup.

    b. Change the bit maps to indicate that space has been assigned to the backup file.

    c. Write the adjusted in-memory FDR to disk to replace the skeletal FDR previously placed there.

    d. Restore the volume name from $$$$$$$$ to its original value.

For tapes, cleanup consists of the following:

* Write a double EOF mark to the tape.
* Position the tape to just before or just after the EOF marks.

12. PUT FINAL STATISTICS AND FINISH TIME TO THE INTERACTIVE AND LISTING DEVICES

13. CLOSE LUNOS USED FOR THIS RERUN.

    The LUNOs used for each rerun are closed from the main routine, BDD, at the conclusion of the rerun.

14. RELEASE LUNOS BEFORE EXITING.

    Release of the LUNOs used by BDD is performed by BDD's substitute S$TERM routine. This substitute routine is always called to terminate the BDD program provided that the system disk drive was not involved in any of the reruns. If the system disk was involved, BDD does not release LUNOs, but performs a LIMI 0 and forces the user to boot the system.

STEPS	
9-10	Are repeated until the source directory is exhausted.
2-13	Are repeated for each rerun.
4-11	Are repeated for second pass tape verification, but instead of writing the destination, the tape is read and compared to the I/O buffer which would have been written.

13.4.3 BDD Modules.

The following discussion of BDD modules explains the work done by each routine.

13.4.3.1 BDABSQ.

This routine produces the abort sequencing messages for various fatal errors encountered during processing. It can abort the current rerun by setting the global common flag ABORT_RUN.

13.4.3.2 BDADJF.

This routine is called when a directory sector is encountered in the FDR buffer. BDADJF adjusts the FDR buffer by moving the next SECT_PER_LEVEL sectors up in the FDR buffer, so that they reside on level TOP_Q (instead of spanning other lower levels as they do when they are read into the buffer).

13.4.3.3 BDADKD.

This routine reads and moves either a KDR or all the ADRs for a file. The KDR or the first alias resides at SECOFF sectors from the beginning of the directory containing this file. The flag MOVKDR indicates whether the KDR or a chain of ADRs is being moved. Channels (DNOS only) are handled in an identical manner to aliases.

13.4.3.4 BDAPLF.

This routine appends the leaf component, LEAF, to the already constructed directory part of the pathname DPTH. It is used to construct pathnames to dump for the user when errors occur.

13.4.3.5 BDBFDR.

This routine builds a rudimentary FDR for the backup file. (It should be called only if DEST_IS_DISK is true.)

13.4.3.6 BDBFSZ.

This routine calculates the buffer sizes and sets pointers to the buffers based on the sector length of the source (and the destination if disk), and the amount of free space (calculated in BDGPRM).

13.4.3.7 BDBHED.

This routine builds a header record for the first volume of the backup file.

13.4.3.8 BDBKUP.

This is the driver routine for each BDD rerun.

13.4.3.9 BDC2NM.

This routine converts an 8 byte character string to the type NAME, and returns this name variable in OUTNAME.

13.4.3.10 BDCHBM.

This routine changes the partial bit maps on disk, beginning at the bit that represents STRT_ADU and continuing for a length of LENGTH ADUs (bits). If ALLOCATE is true, the routine allocates these ADUS (sets bits to 1). If ALLOCATE is not true, the routine releases these ADUs (sets bits to 0).

13.4.3.11 BDCKRD.

This routine checks on the read status of the FDR buffer.

13.4.3.12 BDCKVR.

This routine checks to see if a verification read has completed. If so, it sets the common flag VERIF_READY to true.

13.4.3.13 BDCLCT.

This routine collects the volume information for either the source (SRC) or destination (DST) disk from LOCALSEC.

13.4.3.14 BDCLNP.

This routine performs the final clean-up operations after a rerun has completed. This routine's responsibilities are:

For disks:

1. Make the in-memory copy of the backup file FDR reflect the amount of disk space actually used by the file.

2. Make the bit maps on the destination disk reflect the amount of disk space actually used by the backup file.

3. Make the disk copy of the backup file FDR the same as the in-memory copy (after modified by Step 1).

4. Restore the volume name from $$$$$$$$ to the original value.

For tapes:

1. Write a double EOF mark on the tape.

2. Position the tape to just before or just after the double EOF.

13.4.3.15 BDD.

This routine is the entry point for task BDD.

13.4.3.16 BDD1.

This routine is called by the main routine, BDD, to repeat the backup logic for the second pass against tapes.

13.4.3.17 BDDIRT.

This routine adds a directory pathname to the data structure used for tracing pathnames in the write (I/O) buffers.

13.4.3.18 BDDOND.

For disks only, this routine checks to see if any of the destination IRBs have completed an outstanding write. Screen statistics may also be printed from this module when a completed write is detected and the screen has not been updated for more than the threshold number of times (NEWSCREEN = threshold trigger).

13.4.3.19 BDDONT.

For tapes only, this routine checks to see if any of the destination IRBs have completed an outstanding write on the write buffer. If so, it reports any errors and marks the IRB as free. The I/O buffers are searched in the order they were written (tracked by WB_TAPE_CNT). Screen statistics may be printed from this module when a completed write (in the first pass) or read (in the second pass) is detected and the screen has not been updated for more than the threshold number of times (NEWSCREEN = threshold trigger).

Two explicit cases are handled:

* First pass (Backup Pass)

* S2ND_PASS (Verification Pass)

During the first pass, if an end-of-tape marker is detected, control is passed to routine BDNXVL to handle the transition to the next destination volume. During second pass verification, volume switch time is detected by the trailer record on the tape. BDNXVL is still called to switch volumes, but control returns to BDDONT to complete processing after the volume switch.

13.4.3.20 BDDUMP.

This routine constructs the pathnames of all files in the I/O buffer WICH, and dumps them (print to screen) for the user. It does this based on the trace data structures.

13.4.3.21 BDFDRF.

This routine processes a file FDR for the routine BDFDRP. The flow of control continues from this module through the call to BDMVIO. BDFDRF is called when BDFDRP has detected a file FDR in the FDR buffer.

13.4.3.22 BDFDRP.

This routine is the driver to process the FDR buffer.

13.4.3.23 BDFILT.

This routine adds the leaf component (name) of a file to the trace data structure for tracing file names in the write (I/O) buffers.

13.4.3.24 BDFIND.

This routine finds the disk address of the top level directory or file pathname to backup (specified to the initial task by the user). There are really three cases :

* The user is backing up a normal directory.

* The user is backing up the VCATALOG directory (whole volume).

* The user is backing up a single file.

13.4.3.25 BDFIXD.

This special purpose routine unconditionally restores the volume name of the destination disk to the name specified in PATHS[DPATH]. Also, if the parameter HKC is true, the routine decrements the FDRHKC in the FDR residing at <STRT_ADU,STRT_SEC>, which is presumably the disk address of where the backup file name hashes in VCATALOG on the destination.

13.4.3.26 BDFLBL.

This routine performs a back-space/forward-space LUNO against the LUNO for the destination device (the flag FORWARD determines which). It returns any error code received from this operation in ERRCOD.

13.4.3.27 BDFLSH.

This routine will be called once for each rerun to flush any unwritten or partially filled I/O buffers to the destination.

13.4.3.28 BDGBLK.

This routine obtains a source (or destination) dedicated IRB and returns it to the caller, or returns an invalid index into the array of IRBs (= 0) if all the IRBs are currently in use.

13.4.3.29 BDGPRM.

This routine gets the bid parameters from the file .S$BDD that define the execution parameters for BDD. This procedure is called exactly once when the initial task transfers control to BDD.

It also gets the memory needed to trace pathnames in the write buffers and assigns to the memory obtained for such purpose the pointers LEAFS and DIREC, for leaf names and directory names respectively.

13.4.3.30 BDGTIM.

This routine performs a Get Date and Time SVC and puts the five words returned from this SVC in the time buffer, TIME_BUFF.

13.4.3.31 BDGVIF.

This routine gets the appropriate volume information for the source disk from ADU 0, sector 0. It saves the pertinent information in global variables for future reference during this rerun of BDD.

13.4.3.32 BDHASH.

This routine hashes NAME2H into a directory of length "DIRLEN". It assigns the hash key value of the name to this function.

13.4.3.33 BDINCM.

This routine performs data initializations for common variables.

13.4.3.34 BDINIO.

This routine sets the I/O buffer control variables to their initialized state.

13.4.3.35 BDINVF.

This routine initializes the common variables used for verification.

13.4.3.36 BDIOPR.

This routine processes the I/O buffers. It acts as the driver for moving data through the I/O buffers. I/O is done in initiate mode.

13.4.3.37 BDIOQT.

This routine forces any I/O started in INITATE mode to complete.

13.4.3.38 BDMESG.

This routine writes messages for the user to the terminal and the listing device. All user seen output comes from this module.

13.4.3.39 BDMONT.

This routine issues a mount volume request for the next volume of a backup. It forces the user to mount the requested next volume or quit the current rerun.

13.4.3.40 BDMPTH.

This routine makes a fully expanded pathname representing the current directory being backed up.

13.4.3.41 BDMVIO.

This routine moves data from either the FDR buffer (if FROMFDR = true), or a READ buffer to the next available spots in the current I/O buffer. It is the task standard way to move data into an I/O buffer.

13.4.3.42 BDMVPR.

This routine moves the physical records of the file being backed up to an I/O buffer. This is a <u>speed critical</u> module. Care has been taken to avoid unneeded calls to the Pascal run time. (The source has been written to force Rifle to generate the desired code.) Any added Pascal run-time calls will have a direct negative impact on task speed. This module is the work-horse for the backup process, and is the most important module in the task.

13.4.3.43 BDNMEQ.

This routine determines whether two NAME variables are equal.

13.4.3.44 BDNXVL.

This routine controls volume switches for BDD. It is called from BDWDSK for disks, and BDDONT for tapes.

13.4.3.45 BDOPEN.

This routine opens the devices and returns errors.

13.4.3.46 BDPFDR.

This routine puts the backup file FDR on the destination disk.

13.4.3.47 BDPHED.

This routine puts the backup header record on the destination.

13.4.3.48 BDPPTH.

This routine returns a bad pathname to the user.

13.4.3.49 BDPTIM.

This routine puts the begin and end time of each backup to the listing devices. It also paints the VDT screen template the first time.

13.4.3.50 BDQERR.

This routine is the utility to issue an error message (quickly). This routine is a shorthand way of calling BDSVCE to post an SVC error to the user (and append some text), setting the abort flag to inform BDD that the current rerun is aborting, and finally aborting the current rerun by escaping several levels up to the routine BDBKUP.

13.4.3.51 BDREDD.

This routine reads part of a directory into the FDR buffer.

13.4.3.52 BDREDF.

This routine starts a read into the current read buffer (if needed), or sets the read buffer control variables to indicate completion when a read has completed. This module is speed critical. Additional Pascal run-time calls should be avoided.

13.4.3.53 BDREDV.

This routine attempts to start a read into the verify buffer for disk destinations.

13.4.3.54 BDSCAN.

This routine scans the destination disk's partial bit maps to set up an array of the 17 largest free areas (of a minimum size) on the destination disk.

13.4.3.55 BDSCRM.

This routine calculates the number of sectors that still have not been read into the FDR buffer. It makes this calculation for the directory represented by CURDIR[INDEX]. It returns the number of unread sectors in REMAINING.

13.4.3.56 BDSCTY.

This routine finds out what type of sector is at FDRB.LVL[TOP_Q,Q_HEADS[TOP_Q]].

13.4.3.57 BDSORT.

This routine sorts the allocation array produced by BDSCAN in order of allocation size. It puts the sorted results in the common DESTFDR, the in-memory copy of the destination file FDR.

13.4.3.58 BDSPLT.

This routine finds the name for the "CNUMth" component of the pathname stored in PTH. It returns this value in the parameter CNAM (component name) .

13.4.3.59 BDSRCH.

This routine searches the ADUs in a disk's partial bit map, looking for the next block of unallocated ADUs.

13.4.3.60 BDSTBD.

This routine sets the bid parameters for this rerun.

13.4.3.61 BDSVCE.

This routine outputs a message about an SVC error to the user. If the common variable UCODE is non-zero, The routine issues a utility error message for further explanation for the user.

13.4.3.62 BDSVFD.

This routine saves the needed portions of the FDR for a file in other named variables and commons for easy access. It is called by BDFDRF.

13.4.3.63 BDVECT.

This routine sets up the common variable SECVEC for use when unbuffering physical records from source ADU images. This vector (array) is only used if the physical records fit in a single source ADU.

13.4.3.64 BDVERF.

This routine performs verification. It verifies the write buffer specified by V_WICH with the current contents of the verify buffer.

13.4.3.65 BDWBGN.

This routine determines if a write has begun.

13.4.3.66 BDWDSK.

This routine writes backup data to a disk destination.

13.4.3.67 BDWNDX.

This routine computes the word index of FDRB[LEVEL,SECT].

13.4.3.68 BDWRV.

This routine is BDD's write, read, and verify utility. It is used only during volume switches.

13.4.3.69 BDWTAP.

This routine writes backup data to a tape destination.

13.4.3.70 BDZIRB.

This routine zeros an IRB.

13.4.4 BDD Debug Suggestions.

The module BDFDRP is a large case statement with the case selector being the type of sector in the FDR buffer to be examined next. This module is a good place to set a breakpoint to find errors on a particular file. Set the breakpoint at the FILSEC case and wait for the file in error to come through. When the FDR for the desired file is seen, follow the control flow from there.

Tape verification errors occur only during second pass processing. Therefore, a good place to begin looking for this type of error is in the module BDD1, since this module is called to repeat the backup logic for the second pass.

Until almost all processing is complete, the FDR on the disk destination is inaccurate. The proper FDR to examine when debugging is the in-memory copy, DESTFDR.

If errors occur on multiple volume backups immediately after switching volumes, the routine BDNXVL should be examined. This module completely controls the switch to a new volume. BDNXVL is called from BDWDSK for disk backups, and BDDONT for tape backups.

For major design changes in which several modules must be compiled often, it is much quicker to use the Pascal configuration processor than to always compile each module separately. The following file can be used as input to the configurator to allow compilation of several modules at one time. This file defines the static nesting structure of the BDD task.

```
*BUILD PROCESS
*LIBRARY S
*ALTOBJ  O
*ADD    BDD
*ADD    BDD    : BDABSQ
*ADD    BDD    : BDFILT
*ADD    BDD    : BDAPLF
*ADD    BDD    : BDBKUP
*ADD    BDBKUP : BDADJF
*ADD    BDBKUP : BDADKD
*ADD    BDBKUP : BDDIRT
*ADD    BDD    : BDBFDR
*ADD    BDBKUP : BDCKRD
*ADD    BDBKUP : BDCKVR
*ADD    BDD    : BDCHBM
*ADD    BDBKUP : BDDOND
*ADD    BDBKUP : BDDONT
*ADD    BDBKUP : BDCLNP
*ADD    BDBKUP : BDQERR
*ADD    BDBKUP : BDFDRP
*ADD    BDBKUP : BDFDRF
*ADD    BDBKUP : BDFLSH
*ADD    BDBKUP : BDIOPR
*ADD    BDBKUP : BDIOQT
*ADD    BDD    : BDMONT
*ADD    BDBKUP : BDMVIO
*ADD    BDBKUP : BDMVPR
*ADD    BDBKUP : BDNXVL
*ADD    BDD    : BDPFDR
*ADD    BDBKUP : BDREDF
*ADD    BDBKUP : BDREDD
*ADD    BDBKUP : BDREDV
*ADD    BDBKUP : BDSVFD
*ADD    BDBKUP : BDSCTY
*ADD    BDD    : BDSCAN
```

```
*ADD   BDBKUP : BDVECT
*ADD   BDD    : BDSRCH
*ADD   BDD    : BDSORT
*ADD   BDBKUP : BDVERF
*ADD   BDBKUP : BDWDSK
*ADD   BDBKUP : BDWTAP
*ADD   BDD    : BDBFSZ
*ADD   BDD    : BDBHED
*ADD   BDD    : BDC2NM
*ADD   BDD    : BDD1
*ADD   BDD    : BDZIRB
*ADD   BDD    : BDSPLT
*ADD   BDD    : BDDUMP
*ADD   BDD    : BDSVCE
*ADD   BDD    : BDFIND
*ADD   BDD    : BDFIXD
*ADD   BDD    : BDFLBL
*ADD   BDD    : BDGTIM
*ADD   BDD    : BDGBLK
*ADD   BDD    : BDGVIF
*ADD   BDGVIF : BDCLCT
*ADD   BDD    : BDGPRM
*ADD   BDD    : BDHASH
*ADD   BDD    : BDINCM
*ADD   BDD    : BDINIO
*ADD   BDD    : BDINVF
*ADD   BDD    : BDMPTH
*ADD   BDD    : BDMESG
*ADD   BDD    : BDNMEQ
*ADD   BDD    : BDOPEN
*ADD   BDD    : BDPHED
*ADD   BDD    : BDPTIM
*ADD   BDD    : BDPPTH
*ADD   BDD    : BDSCRM
*ADD   BDD    : BDSTBD
*ADD   BDD    : BDWNDX
*ADD   BDD    : BDWBGN
*ADD   BDD    : BDWRV
*CAT   PROCESS<O,FIG>
```

13.4.5 Miscellaneous Comments.

BDD uses several assembly language substitute run-time routines. The routines for which there are BDD substitutes are R$ABND, R$EXIT, R$GSHP, R$PBVT, and S$TERM.

Substitute routines are used to save task space for run-time functions which need not be performed for BDD. Refer to the Pascal documentation for a description of the functions performed by each of these routines in the normal Pascal task.

The message texts used by BDD are located in the assembly language module BDTEXT. All text, with three exceptions, is located in this module. The three exceptions are for the end-action message, run-time error message, and the IPL-sequence-required messages. The text for these messages is located in the run-time substitute module R$EXIT. If you change the text of a message (for internationalization for example), note that the length of the message is significant.

The following common variables used by BDD are referenced by one or more BDD assembly language modules: USESYS, GIRB, PRMPTR, and TRERUN.

13.5 SURFACE ANALYSIS ALGORITHM

The following paragraphs contain the metacode and data structures used by IDS and tape build for surface analysis. The Disk Information Table, the master list of all disk information (words per track, default values, pointers, and so on) for each disk type supported by Texas Instruments, can be found in DSC.IDS.SOURCE.DITDAT, where DSC is the DNOS source directory.

13.5.1 Metacode.

```
-----------------------------------------------------------------
BEGIN IDSADS
-----------------------------------------------------------------
This routine will analyze the surface of the disk according
to the options specified in the parameter block.
No surface analysis is performed if the ignore bad tracks
option or restore bad tracks option is specified.


  INPUT:  R3 contains the pointer to a parameter block (IDSPRM)
          indicating the options selected and disk information.

  OUTPUT: BADTAB - The bad tracks found, restored, or deleted are
          combined with the bad track information in the
          bad track table when this routine is called.
          Track 0 sector 1 of the disk is updated.
          The diagnostic track is updated.
-----------------------------------------------------------------
   * Initialize global variables, values dependent on disk type,
   * and values dependent on options specified in IDSPRM.

   CALL IDSIRP(R3)

   IF IDSPRM.FLAGS.DELETE
   THEN
      *Delete bad tracks is specified.
      CALL DELETE_BAD_TRACKS
      ESCAPE TO EXIT999
   ENDIF

   IF IDSPRM.FLAGS.RESTORE
   THEN
      *Restore bad tracks is specified.
      CALL RESTORE_BAD_TRACKS
      ESCAPE TO EXIT999
   ENDIF

   * Initialize graph.
   IF IDSPRM.FLAGS.GRAPH
   THEN
      CALL INITIALIZE_GRAPH
      IF ERROR
      THEN
         GRAPH_FLAG = NO
      ELSE
         GRAPH_FLAG = YES
      ENDIF
   ELSE
      GRAPH_FLAG = NO
   ENDIF

   IF IDSPRM.STARTING_CYLINDER = 0
   THEN
      * This is not an IDS continuation, so read
      * the diagnostic track into the bad track table.
      CALL READ_DIAGNOSTIC_TRACK
      IGNORE ERRORS
    ENDIF

   * Analyze each cylinder on the disk.
   FOR CYLINDER = IDSPRM.STARTING_CYLINDER TO TOTAL_CYLINDERS - 1
      * Since some disks return errors if cylinder is changed
      * with offsets active, issue a seek for this cylinder.
      * If disk type supports offsets, issue restore command.
      ISSUE SEEK DIRECT TILINE DISK COMMAND
      IF SA_ASSIST = YES
      THEN
         CLEAR REC_DEFECT_LEN table
      ENDIF
      CALL ANALYZE_CYLINDER
      IF FATAL ERROR: EXIT1
      CALL UPDATE_AFTER_CYLINDER
      IF FATAL ERROR: EXIT1
   CONTINUE

   * Sort the bad track table.
   CALL  IDSSRT(@BADTAB)

   * Map bad tracks.
   CALL MAP_BAD_TRACKS

   * Write the diagnostic track.
   CALL WRITE_DIAGNOSTIC_TRACK
   IF FATAL ERROR: EXIT1

   * Write the bad track list on track 0, sector 1.
   CALL AVOID_BAD_TRACKS
   IF FATAL ERROR: EXIT1

   * Note that there is a hole here: the bad track list
   * is in the state 2 format on track 0, sector 1, but
   * the disk is in state 1.

   * Change the disk state to 2.
   CLEAR SECTOR_BUFFER
   SECTOR_BUFFER.SCOSTA = 2
   WRITE SECTOR_BUFFER
   IF FATAL ERROR: EXIT1

   TERMINATE GRAPH DISPLAY

   WRITE SYSTEM LOG MESSAGE FOR NORMAL TERMINATION

   B  EXIT999

----------- ERROR EXIT -----------------

 EXIT1   TERMINATE GRAPH DISPLAY
         WRITE SYSTEM LOG MESSAGE FOR ABNORMAL TERMINATION
         B  EXIT999

 EXIT999 RETURN

----------------------------------------------------------------
END IDSADS
----------------------------------------------------------------
```

```
-------------------------------------------------------------
BEGIN IDSIRP - Initialize Routine Parameters
-------------------------------------------------------------
This routine will initialize global variables that
depend on disk type and options specified in IDSPRM.
INPUT: R3 contains a pointer to parameter block (IDSPRM).
OUTPUT: Global variables initialized.


  @IDSPRM =  R3

  @BADTAB =  @IDSPRM.BADTRK

  * READ_TYPES_WORD has bits set corresponding to
  * each read type used for surface analysis.
  READ_TYPES_WORD = IDSPRM.DITPTR.DITRTF

  NUM_RD_TYPES = number of bits set in READ_TYPES_WORD


  IF IDSPRM.TSTLVL .EQ. 'L'
    NUM_PATTERNS=4
    WRTFMTS_PER_PATTERN=8
  ELSE

    IF IDSPRM.TSTLVL .EQ. 'M'
      NUM_PATTERNS=4
      WRTFMTS_PER_PATTERN=5
    ELSE

      IF IDSPRM.TSTLVL .EQ. 'S'
        NUM_PATTERNS=2
        WRTFMTS_PER_PATTERN=2
      ENDIF
    ENDIF
  ENDIF

  IF READ_TYPES_WORD indicates only nominal read
  THEN
    WRTFMTS_PER_PATTERN= WRTFMTS_PER_PATTERN * 4
  ENDIF

  PATTERNS(1) = @IDSPRM.DIT.@PATTERNS(1)
  PATTERNS(2) = @IDSPRM.DIT.@PATTERNS(2)
  IF @IDSPRM.UP1 = 0
  THEN
     * User did not enter patterns.
     PATTERNS(3) = IDSPRM.DIT.@PATTERNS(3)
     PATTERNS(4) = IDSPRM.DIT.@PATTERNS(4)
  ELSE
     * User has entered patterns.
     PATTERNS(3) = @IDSPRM.UP1
     PATTERNS(4) = @IDSPRM.UP2
```

```
 ENDIF

  * The legal limit of the bad track table is calculated
  * based on the disk type.  64 bad tracks are allowed for
  * all disks. The number of additional tracks allowed for
  * disks with bad track mapping depends on the number of
  * spare tracks available for mapping.

  CALCULATE BAD_TRACK_TABLE_LEGAL_LIMIT based on disk type

  * Note: Some disks support surface analysis assistance
  * (SA-Assist) only if they have a controller of a certain
  * revision level. Other disks support SA-Assist regardless
  * of the controller revision level.
  * To determine the revision level of the controller,
  * IDS issues a self test command >7C.

  IF DITDAT.SA_ASSIST .EQ. YES
  THEN
     IF DITDAT.ISSUE_SELF_TEST .EQ.YES
     THEN
        CLEAR SELF TEST RETURN BUFFER
        ISSUE SELF TEST >7C
        IF REVISION OF CONTROLER .GE. TABLE_VALUE
        THEN
          SA_ASSIST = YES
        ELSE
           SA_ASSIST = NO
        ENDIF
     ELSE
        SA_ASSIST = YES
     ENDIF
  ELSE
     SA_ASSIST = NO
  ENDIF


TOTAL_HEADS =
          FIRST FIVE BITS OF WORD 3 OF STORE REGISTERS FROM DIT
TOTAL_CYLINDERS =
          LAST 11 BITS OF WORD 3 OF STORE REGISTERS FROM DIT

 * The recorded defect length table is initialized to 2 in every
 * entry. It is important to note that if SA-Assist is supported,
 * the recorded defect length will be zeroed before the analysis
 * of each cylinder.  If SA-Assist is not supported, the recorded
 * defect length remains set to 2 and is never changed.
 INITIALIZE REC_DEFECT_LENGTH table TO 2 IN EVERY ENTRY

 IF @IDSPRM.FLAGS.MARK_MARGINAL_TRACKS
 THEN
    ALLOWABLE_TRACK_PERCENTAGE = 100
 ELSE
```

```
      ALLOWABLE_TRACK_PERCENTAGE = 300
   ENDIF

   IF @IDSPRM.DIT.TRANSFER INHIBIT FLAG = NO
   THEN
      SET UP GLOBAL VALUES USED TO FAKE TRANSFER INHIBIT
      REFORMAT_FLAG = YES
   ELSE
      SET UP GLOBAL VALUES TO USE WITH TRANSFER INHIBIT
      REFORMAT_FLAG = NO
   ENDIF

   SET UP GLOBAL PARAMETERS THAT DEPEND ON DISK TYPE FOR
   ISSUE_TILINE_DISK_COMMAND ROUTINE


-----------------------------------------------------------------
END IDSIRP
-----------------------------------------------------------------
```

```
----------------------------------------------------------------
BEGIN ANALYZE_CYLINDER
----------------------------------------------------------------
This routine will completely analyze one cylinder of a disk.
The cylinder is input by the value of R1.

  GLOBAL DATA RELEVANT ON INPUT
          SA_ASSIST
          TOTAL_HEADS
          WRTFMTS_PER_PATTERN
          NUM_PATTERNS

  OUTPUT: The cylinder specified by R1 is analyzed.

          BAD_RDS_PER_PATTERN table contains the count of bad
          reads per pattern in the entry corresponding to
          each pattern and each head.

          TOT_RDS_PER_PATTERN table contains the total reads
          per pattern in the entry corresponding to each pattern
          for each head.

          HEAD_ERROR_FLAG table contains an entry for each head:

           - An entry of 0 indicates the head never had an error.

           - An entry of 1 indicates the head had an error, but
             it has not been determined to be bad.

           - An entry of -1 indicates the track corresponding
             to that head has been determined to be bad.

----------------------------------------------------------------

   CLEAR BAD_RDS_PATTERN table
   CLEAR TOT_RDS_PATTERN table
   CLEAR HEAD_ERROR_FLAG table
   *
   * This loop will repeat only once unless there is an error.
   * If there is an error, the loop will repeat five times to
   * perform "overdrive" on each head with an error.
   * Overdrive is caused by REPETITIONS being set to 5.  This
   * occurs whenever an error occurs in FORMAT_AND_READ_TRACK
   * or SA_ASSIST_FORMAT_AND_READ_CYLINDER.

   REPETITIONS = 1
   FOR  COUNT = 1 TO REPETITIONS

      FOR  PATRN_INDX = 1 TO NUM_PATTERNS

         FOR FORMAT_COUNT = 1 TO WRTFMTS_PER_PATTERN
            SET FORMAT PATTERN IN BUFFER
            IF COUNT = 1
          THEN
             * The first time through the outer loop, count
             * is 1, so analyze every track on the cylinder.
             IF SA_ASSIST = YES
             THEN
                CALL SA_ASSIST_FORMAT_AND_READ_CYLINDER
                IF FATAL_ERROR: ESCAPE, ERROR CODE IN RO
             ELSE
                CALL FORMAT_AND_READ_CYLINDER
                IF FATAL_ERROR: ESCAPE, ERROR CODE IN RO
             ENDIF
          ELSE
             * If in overdrive (count is not 1), analyze only
             * those tracks on the cylinder which had errors.
             * These tracks are indicated by a 1
             * in HEAD_ERROR_FLAG.
             FOR HEAD = 1 TO TOTAL_HEADS
                IF HEAD_ERROR_FLAG(HEAD) = 1
                THEN
                   IF SA_ASSIST = YES
                   THEN
                      CALL SA_ASSIST_FORMAT_AND_READ_TRACK
                      IF FATAL_ERROR: ESCAPE, ERROR CODE IN RO
                   ELSE
                      CALL FORMAT_AND_READ_TRACK
                      IF FATAL_ERROR: ESCAPE, ERROR CODE IN RO
                   ENDIF
                ENDIF
             CONTINUE (for head = 1 to total_heads)
          ENDIF
       CONTINUE (for format_count = 1 to wrtfmts_per_pattern)
    CONTINUE (for  patrn_indx = 1 to num_patterns)
 CONTINUE (for count = 1 to repetition)
RETURN
-----------------------------------------------------------------
END ANALYZE_CYLINDER
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN UPDATE_AFTER_CYLINDER
-----------------------------------------------------------------
This routine is called after each cylinder is analyzed.
It will call ANALYZE_TRACK_INFORMATION to evaluate the track.
It will also update the bad track table, update state 1 and graph
display if necessary, and reformat the track if necessary.

GLOBAL DATA RELEVANT ON INPUT
   CURGRAPH
   REFORMAT_FLAG
   TOTAL_CYLINDERS

GLOBAL DATA RELEVANT ON OUTPUT
   CURGRAPH
-----------------------------------------------------------------

 IF REFORMAT_FLAG = YES
 THEN
    CALL REFORMAT_CYLINDER
 ENDIF

 CALL ANALYZE_TRACK_INFORMATION

 CALL UPDATE_BAD_TRACK_TABLE
 IF FATAL ERROR: ESCAPE WITH ERROR CODE IN R0

 INCREMENT = (TOTAL_CYLINDERS * 79) / CYLINDER
 IF INCREMENT .GT. CURGRAPH or CYLINDER = 0
 THEN
    CALL UPDATE_STATE1
    CURGRAPH = INCREMENT
    IF GRAPH_FLAG = YES
    THEN
       CALL IDSDSP(INCREMENT)
    ENDIF
  ENDIF
-----------------------------------------------------------------
END UPDATE_AFTER_CYLINDER
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN FORMAT_AND_READ_CYLINDER
-----------------------------------------------------------------
This routine is called only if SA_ASSIST = NO.
This routine will call FORMAT_AND_READ_TRACK for each head on the
cylinder.

GLOBAL DATA RELEVANT ON INPUT
        TOTAL_HEADS

GLOBAL DATA RELEVANT ON OUTPUT
        BAD_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
        TOT_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
        HEAD_ERROR_FLAG(HEAD) is updated.
        REPETITIONS is set to 5 if any errors are encountered.
-----------------------------------------------------------------


   FOR HEAD = 1 TO TOTAL_HEADS
      IF HEAD_ERROR_FLAG(HEAD) .NE. >FF
      THEN
         CALL FORMAT_AND_READ_TRACK
         IF FATAL_ERROR: ESCAPE WITH ERROR CODE IN R0
      ENDIF
   CONTINUE
   RETURN
-----------------------------------------------------------------
END FORMAT_AND_READ_CYLINDER
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN FORMAT_AND_READ_TRACK
-----------------------------------------------------------------
This routine is called only if SA_ASSIST = NO.
It performs a write format operation with the pattern indicated
by PTRN_INDX on the track indicated by CYLINDER and HEAD.
After the write, a read will be issued for each read type.
If any read type fails, the failing read type will be read
nine more times. Read types which have not failed will be
read four more times. If any errors are found, REPETITIONS
will be set to 5 to force overdrive in the outermost loop of
ANALYZE_CYLINDER, and HEAD_ERROR_FLAG will be set to 1.

GLOBAL DATA RELEVANT ON INPUT
       CYLNDER - Cylinder containing track to analyze.
       PTRN_INDX - Index of pattern to use.
       HEAD - Head number to analyze.
       READ_TYPES_WORD

GLOBAL DATA RELEVANT ON OUTPUT
       BAD_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
       TOT_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
       HEAD_ERROR_FLAG(HEAD) is updated.
       REPETITIONS is set to 5 if any errors are encountered.
-----------------------------------------------------------------
   ISSUE WRITE FORMAT WITH PATTERN
   IGNORE ERRORS

   CLEAR FAILURES_PER_READ_TYPE table
   BAD_RDS_WRTFMT= 0
   TOT_RDS_WRTFMT = 0
   CLEAR MEDIA_ERROR_FLAG

   * Issue a read for each read type.
   FOR I = 1 TO 14
      IF READ_TYPES_WORD(bit I) = 1
      THEN
         SET READ_TYPE(I) IN >18 SVC BLOCK
         SET READ COMMAND IN >18 SVC BLOCK
         BL ISSUE_TILINE_DISK_COMMAND

         TOT_RDS_WRTFMT = TOT_RDS_WRTFMT + 1

         IF ERROR
         THEN
            IF ERROR .EQ. MEDIA ERROR
            THEN
               FAILURES_PER_READ_TYPE(I) = 1
               SET MEDIA_ERROR_FLAG
               REPETITION = 5
               HEAD_ERROR_FLAG(HEAD) = 1
            ELSE
               ESCAPE FORMAT_AND_READ_TRACK WITH ERROR CODE IN R0
            ENDIF
         ENDIF
      ENDIF

   CONTINUE

   IF MEDIA_ERROR_FLAG SET
   THEN
      * Got an error on this track, so read it nine more
      * times for every read type that failed and four more
      * times for every read type without errors.

      FOR I = 1 TO 14
         IF READ_TYPES_WORD(bit I) = 1
         THEN

           FOR N = 1 TO 9

              * Only do read if read type has failed or if the
              * track has not already been read four more times.
              IF N LT. 5 .OR. FAILURES_PER_READ_TYPE(I) .NE. ZERO
              THEN
                 SET READ_TYPE(I) IN SVC >18 CALL BLOCK
                 BL ISSUE_TILINE_DISK_COMMAND
                 TOT_RDS_WRTFMT = TOT_RDS_WRTFMT + 1

                 IF ERROR
                 THEN
                    IF ERROR .EQ. MEDIA ERROR
                    THEN
                       INCREMENT FAILURES_PER_READ_TYPE(I)
                       IF FAILURES_PER_READ_TYPE(I) .GE. 6 THEN
                       THEN
                    * Track is bad: 60% of reads failed for type I.
                          HEAD_ERROR_FLAG(HEAD) = >FF
                          ESCAPE FORMAT_AND_READ_TRACK
                       ENDIF
                    ENDIF (if media error)
                    ELSE
                       ESCAPE FORMAT_AND_READ_TRACK
                 ENDIF (if error)
              ENDIF (if failure or n < 5)
        CONTINUE
        BAD_RDS_WRTFMT= BAD_RDS_WRTFMT + FAILURES_PER_READ_TYPE(I)
      ENDIF
    CONTINUE
  ENDIF
  CALL ANALYZE_FORMAT_INFORMATION
  RETURN
-------------------------------------------------------------------
END FORMAT_AND_READ_TRACK
-------------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN SA_ASSIST_FORMAT_AND_READ_CYLINDER
-----------------------------------------------------------------
This routine is called only if SA_ASSIST = YES.
This routine performs a write format operation using the pattern
indicated by PTRN_INDX on the cylinder indicated by CYLNDR.
It will then call SA_ASSIST_STATISTICS to perform the additional
reads if any errors occur.

GLOBAL DATA RELEVANT ON INPUT
       CYLINDER - Cylinder number to format and analyze.
       PTRN_INDX - Index of pattern to use.
       NUM_RD_TYPES
       READ_TYPES_WORD
       TOTAL_HEADS

GLOBAL DATA RELEVANT ON OUTPUT
       BAD_RDS_PATTERN(HEAD,PTRN_INDX)is updated for each head.
       TOT_RDS_PATTERN(HEAD,PTRN_INDX)is updated for each head.
       HEAD_ERROR_FLAG(HEAD) is updated for all heads.
-----------------------------------------------------------------
  * Set up SA_ASSIST_BUF for write format with pattern.
  SA_ASSIST_BUF.FLAGS =  READ_TYPES_WORD
  SA_ASSIST_BUF.WRITE_FORMAT_FLAG = YES
  SA_ASSIST_BUF.CYLINDER_MODE = YES

  * Issue write format.
  SET HEAD TO ZERO
  BL ISSUE TILINE_DISK_COMMAND

FOR HEAD = 1 TO TOTAL_HEADS
 IF HEAD_ERROR_FLAG(HEAD) .NE. >FF
 THEN
      TOT_RDS_WRTFMT= NUM_RD_TYPES
      BAD_RDS_WRTFMT= 0
      CLEAR FAILURES_PER_READ_TYPE table

* Recorded defect length table is cleared in IDSADS.
 IF SA_ASSIST_BUF.FAILING_READ_TYPE(HEAD) <> ZERO
 THEN
   IF REC_DEFECT_LEN(HEAD) .LE. SA_ASSIST_BUF.DEFECT_LENGTH(HEAD)
   THEN
      REC_DEFECT_LEN(HEAD)= SA_ASSIST_BUF.DEFECT_LENGTH(HEAD)
   ENDIF

   HEAD_ERROR_FLAG(HEAD) = 1
          REPETITIONS = 5

          FOR J = 1 TO 14
             IF SA_ASSIST_BUF.FAILING_READ_TYPES(HEAD,bit J)
             THEN
                INCREMENT FAILURES_PER_READ_TYPE(J)
                BAD_RDS_WRTFMT= BAD_RDS_WRTFMT + 1
                ENDIF

             CONTINUE

     ENDIF
     CALL SA_ASSIST_STATISTICS
    ENDIF
    CONTINUE

    RETURN
    ------------------------------------------------------------------
    END SA_ASSIST_FORMAT_AND_READ_CYLINDER
    ------------------------------------------------------------------
```

```
-----------------------------------------------------------------------
BEGIN SA_ASSIST_FORMAT_AND_READ_TRACK
-----------------------------------------------------------------------
This routine is called only if SA_ASSIST = YES.
This routine performs a write format operation using the pattern
indicated by PTRN_INDX on the track indicated by HEAD and CYLNDR.
This routine is never reached unless there has been an
error on this head previously.

GLOBAL DATA RELEVANT ON INPUT
   CYLINDER - Cylinder containing track to analyze.
   PTRN_INDX - Index of pattern to use.
   HEAD - Head number to analyze.

GLOBAL DATA RELEVANT ON OUTPUT
   BAD_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
   TOT_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
   HEAD_ERROR_FLAG(HEAD) contains >FF if track is determined bad.
-----------------------------------------------------------------------
* Set up SA_ASSIST_BUF for write format with pattern in head mode.
  SA_ASSIST_BUF.FLAGS = READ_TYPES_WORD
  SA_ASSIST_BUF.WRITE_FORMAT_FLAG = YES
  SA_ASSIST_BUF.HEAD_MODE = YES
  * Issue TILINE disk command.
  SET HEAD VALUE
  BL ISSUE_TILINE_DISK_COMMAND

  TOT_RDS_WRTFMT= NUM_RD_TYPES
  BAD_RDS_WRTFMT= 0
  CLEAR FAILURES_PER_READ_TYPE table

  IF SA_ASSIST_BUF.FAILING_READ_TYPE(1) <> ZERO
  THEN
     IF REC_DEFECT_LEN(HEAD) .LE. SA_ASSIST_BUF.DEFECT_LENGTH(1)
     THEN
        REC_DEFECT_LEN(HEAD)= SA_ASSIST_BUF.DEFECT_LENGTH(1)
     ENDIF

     FOR J = 1 TO 14 DO
         IF SA_ASSIST_BUF.FAILING_READ_TYPE(1,bit J)
         THEN
           INCREMENT FAILURES_PER_READ_TYPE(J)
           BAD_RDS_WRTFMT= BAD_RDS_WRTFMT + 1
         ENDIF
     CONTINUE
  ENDIF
  BL SA_ASSIST_STATISTICS
RETURN
-----------------------------------------------------------------------
END SA_ASSIST_FORMAT_AND_READ_TRACK
-----------------------------------------------------------------------
```

```
--------------------------------------------------------------------
BEGIN  SA_ASSIST_STATISTICS (SAS)
--------------------------------------------------------------------
This routine will perform the additional reads to gather
statistics for analyzing the specified track (HEAD).
Each read type which fails will be read nine more times.
Read types without errors will be read four more times.

GLOBAL DATA RELEVANT ON INPUT
        HEAD
        BAD_RDS_WRTFMT
        TOT_RDS_WRTFMT
        FAILURES_PER_READ_TYPE table
        NUM_RD_TYPES

GLOBAL DATA RELEVANT ON OUTPUT
        BAD_RDS_WRTFMT is updated.
        TOT_RDS_WRTFMT is updated.
        HEAD_ERROR_FLAG is updated.
--------------------------------------------------------------------

IF BAD_RDS_WRTFMT .NE. ZERO
THEN
  * There has been an error, so statistics will be gathered.

  * Set up SA_ASSIST_BUF for read without format in head mode.
  SET MODE TO READ WITHOUT WRITE FORMAT
  SA_ASSIST_BUF.HEAD_MODE = YES

  * Issue TILINE disk command.
  SET HEAD VALUE

  CUR_NUM_READ_TYPES = NUM_READ_TYPES

  FOR N = 1 TO 9 DO

     BL ISSUE_TILINE_DISK_COMMAND

     TOT_RDS_WRTFMT = TOT_RDS_WRTFMT + CUR_NUM_READ_TYPES

     IF SA_ASSIST_BUF.FAILING_READ_TYPE(1) .NE. ZERO
     THEN

      IF REC_DEFECT_LEN(HEAD) .LE. SA_ASSIST_BUF.DEFECT_LENGTH(1)
      THEN
         REC_DEFECT_LEN(HEAD)= SA_ASSIST_BUF.DEFECT_LENGTH(1)
      ENDIF

      FOR J = 1 TO 14 DO
         IF SA_ASSIST_BUF.FAILING_READ_TYPE(1,bit J)
         THEN
            INCREMENT FAILURES_PER_READ_TYPE(J)
            BAD_RDS_WRTFMT= BAD_RDS_WRTFMT + 1
            IF FAILURES_PER_READ_TYPE(J) .GE. 6
            THEN
               * Track is bad: 60% of reads failed for read type.
               HEAD_ERROR_FLAG(HEAD) = >FF
               ESCAPE SA_ASSIST_STATISTICS
            ENDIF
         ENDIF

      CONTINUE (for j = 1 to 14)

   ENDIF (if sa_assist_buf.failed_read_types)

   * After four additional reads on this track, read types that
   * have not failed are reset in the SA_ASSIST_BUF because
   * they have been read five times.  Only those read types which
   * have failed need to be read ten times.
   IF N .EQ. 4
   THEN
      FOR J = 1 TO 14 DO
         IF FAILURES_PER_READ_TYPE(J) .EQ. 0
         THEN
            SA_ASSIST_BUF.FLAGS(bit J) = 0
            CUR_NUM_READ_TYPES = CUR_NUM_READ_TYPES - 1
         ENDIF
       CONTINUE
   ENDIF

  CONTINUE  (For n = 1 to 9)
ENDIF

BL ANALYZE FORMAT INFORMATION


RETURN
------------------------------------------------------------------
END SA_ASSIST_STATISTICS
------------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN ANALYZE_FORMAT_INFORMATION
-----------------------------------------------------------------
This routine is called after all the reads are complete for a
write format operation.  It verifies that 45% of all reads
for this format did not fail.  It also accumulates the total
reads and bad reads for the track.

GLOBAL DATA RELEVANT ON INPUT
        HEAD - head number analyzed
        BAD_RDS_WRTFMT
        TOT_RDS_WRTFMT
        PTRN_INDX

OUTPUT:
   BAD_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
   TOT_RDS_PATTERN(HEAD,PTRN_INDX) is updated.
   HEAD_ERROR_FLAG(HEAD) - set to >FF if track is determined bad.
-----------------------------------------------------------------


   TOT_RDS_PATTERN(HEAD,PTRN_INDX) =
            TOT_RDS_PATTERN(HEAD,PTRN_INDX) + TOT_RDS_WRTFMT
   IF BAD_RDS_WRTFMT .NE. 0
   THEN

      BAD_RDS_PATTERN(HEAD,PTRN_INDX) =
            BAD_RDS_PATTERN(HEAD,PTRN_INDX) + BAD_RDS_WRTFMT

      BAD_RDS_LIMIT= TOT_RDS_WRTFMT * 45%
      IF BAD_RDS_WRTFMT .GE. BAD_RDS_LIMIT
      THEN
         HEAD_ERROR_FLAG(HEAD) = >FF
      ENDIF
   ENDIF

RETURN
-----------------------------------------------------------------
END ANALYZE_FORMAT_INFORMATION
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN ANALYZE_TRACK_INFORMATION
-----------------------------------------------------------------
This routine is called after each cylinder is analyzed.
The information about bad reads and total reads has been
accumulated for each format pattern in two tables.
This routine analyzes those two tables.

GLOBAL DATA RELEVANT ON INPUT
   TOTAL_HEADS
   NUM_PATTERNS
   BAD_RDS_PATTERN table
   TOT_RDS_PATTERN table

GLOBAL DATA RELEVANT ON OUTPUT
  HEAD_ERROR_FLAG will contain >FF for every track determined bad
-----------------------------------------------------------------

   FOR HEAD = 1 TO TOTAL_HEADS
      BAD_RDS_TRK = 0
      TOT_RDS_TRK = 0

      FOR PTRN_INDX = 1 TO NUM_PATTERNS
         IF BAD_RDS_PATTERN(HEAD,PTRN_INDX) .NE. ZERO
         THEN
            BAD_RDS_LIMIT =
                          TOT_RDS_PATTERN(HEAD,PTRN_INDX) * 37.5%
            IF BAD_RDS_PATTERN(HEAD,PTRN_INDX) .GE. BAD_RDS_LIMIT
            THEN
               HEAD_ERROR_FLAG(HEAD) = >FF
            ENDIF
            BAD_RDS_TRK = BAD_RDS_TRK +
                                  BAD_RDS_PATTRN(HEAD,PTRN_INDX)
         ENDIF
         TOT_RDS_TRK = TOT_RDS_TRK +
                                  TOT_RDS_PATTRN(HEAD,PTRN_INDX)
      CONTINUE

      BAD_RDS_LIMIT =
                  TOT_RDS_TRK * ALLOWABLE_TRACK_PERCENTAGE / 100
      IF BAD_RDS_TRK .GE. BAD_RDS_LIMIT
      THEN
         HEAD_ERROR_FLAG(HEAD) = >FF
      ENDIF
   CONTINUE


RETURN
-----------------------------------------------------------------
END ANALYZE_TRACK_INFORMATION
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
BEGIN UPDATE_BAD_TRACK_TABLE
-----------------------------------------------------------------
This routine will make an entry in the bad track table
for every bad track found on the cylinder.

GLOBAL DATA RELEVANT ON INPUT
       TOTAL_HEADS
       HEAD_ERROR_FLAG - One entry for each head; >FF in each
                         entry with a bad track.
       CYLNDER  - Cylinder analyzed.
       BADTAB  - Bad track table.

GLOBAL DATA RELEVANT ON OUTPUT
        BADTAB - Bad track table is updated.
-----------------------------------------------------------------
  FOR HEAD = 1 TO TOTAL_HEADS

    IF HEAD_ERROR_FLAG(HEAD) .EQ. >FF
    THEN
       IF CYLINDER = 0 AND (HEAD = 0 OR HEAD = 1)
       THEN
          ATTEMPT TO CLEAR TRACK 0 SECTOR 0 to erase state 1
          CALL ERRINT(ERROR9)
          ESCAPE WITH ERROR CODE IN R0
       ENDIF

       IF BAD_TRACK_TABLE + 2 = BAD_TRACK_TABLE_LEGAL_LIMIT
       * Bad track table full?
       THEN
          * Sort the bad track table.
          CALL ADSSRT
          IF BAD_TRACK_TABLE +2 = BAD_TRACK_TABLE_LEGAL_LIMIT
          * Is the bad track table still full?
          THEN
             CALL ERRINT(ERROR5)
             ESCAPE WITH ERROR CODE IN R0
          ENDIF
       ENDIF
       PLACE CYLNDER, HEAD, AND REC_DEFECT_LENGTH(HEAD) IN THE
       NEXT AVAILABLE ENTRY OF BADTAB

    ENDIF
  CONTINUE

RETURN
-----------------------------------------------------------------
END UPDATE_BAD_TRACK_TABLE
-----------------------------------------------------------------
```

13.5.2 IDS Data Structures.

The following paragraphs describe the IDS data structures.

13.5.2.1 IDSPRM.

A pointer to this structure is in R3 when IDSADS is called.

```
      *----------+----------*
R3->  |  FLAGS   | TSTLVL   |   Flags/Testing level
      *----------+----------*
      |         UP1         |   User entered pattern
      *----------+----------*
      |         UP2         |   User entered pattern
      *----------+----------*
      |  STARTING CYLINDER  |   Starting cylinder
      *----------+----------*
      |     INTERLEAVE      |   User specified interleave
      *----------+----------*
      |       BADTRK        |   Pointer to bad track table
      *----------+----------*
      |       DITPTR        |   Pointer to disk table entry
      *----------+----------*
      |DISK LUNO |          |   Disk LUNO
      *----------+----------*
      |      DISK NAME      |   Pointer to disk name
      *----------+----------*

FLAGS: (X.......)  Graph
       (.X......)  Mark questionable tracks
       (..X.....)  Delete - Delete bad track list
       (...X....)  Restore - Restore bad track list
```

13.5.2.2 BADTAB.

```
        *----------+----------*       *-------------+------------*
        ¦ NEXT EMPTY ENTRY    ¦ ---+  ¦BAD_TRACK_TABLE_LEGAL_LIMIT¦
        *----------+----------*    ¦  *-------------+------------*
   +---¦  END OF TABLE       ¦    ¦   The number of bad tracks   ¦
   ¦    *----------+----------*    ¦   allowed depends on disk    ¦
   ¦    ¦    CYLINDER         ¦    ¦   type.  64 bad tracks can   ¦
   ¦    *----------+----------*    ¦   be avoided on all disks.   ¦
   ¦    ¦  HEAD    ¦ DEF LEN  ¦    ¦   Disks that support bad     ¦
   ¦    *----------+----------*    ¦   track mapping are allowed  ¦
   ¦    /          /          /    ¦   to have all spare tracks   ¦
   ¦    /          /          /    ¦   mapped + 64 tracks avoided.¦
   ¦    /          /          /    ¦   The bad track table legal  ¦
   ¦    *----------+----------*    ¦   limit is calculated in     ¦
   ¦    ¦    CYLINDER         ¦    ¦   IDSIRP and points to the   ¦
   ¦    *----------+----------*    ¦   first entry past the point ¦
   ¦    ¦  HEAD    ¦ DEF LEN  ¦    ¦   where the bad track table  ¦
   ¦    *----------+----------*    ¦   is full for this disk type.¦
   ¦    /          /          /<---+                              ¦
   ¦    /   EMPTY ENTRIES     /                                   ¦
   ¦    /          /          /                                   ¦
   ¦    *- - - - - + - - - - -*                                   ¦
   ¦    ¦ LAST LEGAL ENTRY    ¦                                   ¦
   ¦    *- - - - - + - - - - -*                                   ¦
   ¦    ¦          ¦          ¦                                   ¦
   ¦    *- - - - - + - - - - -*                                   ¦
   ¦    /          /          /<----------------------------------+
   ¦    /          /          /   ------------------------------
   ¦    /          /          /   Additional space necessary to
   ¦    /          /          /   allow for duplicates before
   ¦    /          /          /   the table is coalesced.
   ¦    *- - - - - + - - - - -*   ------------------------------
   ¦    ¦ PHYSICAL LAST ENTRY ¦
   ¦    * - - - - -+ - - - - -*
   ¦    ¦          ¦          ¦
   ¦    * - - - - -+ - - - - -*
   ¦
   +---> First word past end of table.
```

The high order bit of the cylinder entry is set to 1 if the bad track is mapped. This allows the bad track table to be written to disk "as-is" on the diagnostic track.

13.5.2.3 READ_TYPES_WORD.

One bit is set for each read type to use. The bit is initialized in IDSIRP from the DITDAT entry for the disk type.

```
   *----------+----------*
   |   READ_TYPES_WORD   |
   *----------+----------*

0  (X...............) - Offset forward
1  (.X..............) - Offset reverse
2  (..X.............) - Strobe early
3  (...X............) - Strobe late
4  (....X...........) - Nominal
5  (.....X..........) - Strobe early and offset early
6  (......X.........) - Strobe early and offset reverse
7  (.......X........) - Strobe late and offset forward
8  (........X.......) - Strobe late and offset reverse
   (.........XXXXX..) - Spare future read types - zero
   (..............XX) - Never used - zero
```

13.5.2.4 READ_TYPES.

There is room for up to 14 entries corresponding to the read types to be used for surface analysis. The TPCS flags for a read type are contained in the entry corresponding to that read. The read type corresponds to the bit positions defined for READ_TYPES_WORD.

To clarify, READ_TYPES(entry 1) contains the flags for the TPCS to issue the type of read defined for bit 1 of READ_TYPES_WORD.

```
   *----------+----------*
   | MASK FOR READ TYPES |
   *----------+----------*
   /          /          /
   /          /          /
   *----------+----------*
```

13.5.2.5 FAILURES_PER_READ_TYPE.

There is one byte entry for each possible read type (14). Whenever a read type fails, the count is incremented.

This table is initialized after each write format operation.

```
*--------------------*
|  COUNT OF FAILURES |
*--------------------*
/                    /
/                    /
*--------------------*
```

13.5.2.6 SA_ASSIST_BUF.

The first three words of this buffer are set up before issuing the command. The entries corresponding to each head contain information returned from the SA-Assist command. The bits set in FAILING READ TYPES represent which read type has failed. The bits in the flag word and in FAILING READ TYPES are described below.

```
*------------------+--------------------*
|          RESERVED - ALWAYS 0          |
*------------------+--------------------*
|  FLAGS - READ TYPES, CYLINDER, WRITE  |
*------------------+--------------------*
|        FORMAT PATTERN TO USE          |
*------------------+--------------------*  - - - - - - - - -
|          FAILING READ TYPES           |
*------------------+--------------------*   1 entry per head
|  FAILING SECTOR  |   DEFECT LENGTH    |
*------------------+--------------------*  - - - - - - - - -
/                  /                    /
/                  /                    /
*------------------+--------------------*
|          FAILING READ TYPES           |
*------------------+--------------------*
|  FAILING SECTOR  |   DEFECT LENGTH    |
*------------------+--------------------*
```

SA_ASSIST_BUF flags for READ TYPES, CYLINDER, and WRITE:

```
   0  (X...............) - Offset forward
   1  (.X..............) - Offset reverse
   2  (..X.............) - Strobe early
   3  (...X............) - Strobe late
   4  (....X...........) - Nominal
   5  (.....X..........) - Strobe early and offset early
   6  (......X.........) - Strobe early and offset reverse
   7  (.......X........) - Strobe late and offset forward
   8  (........X.......) - Strobe late and offset reverse
   9  (.........XXXXX..) - Spare future read types - zero
   14 (..............X.) - 1 = Write format and read
                         - 0 = Read only
   15 (...............X) - 1 = Issue command for entire cylinder
                         - 0 = Issue command for one head only
```

For FAILING READ TYPES, bits correspond to the flag bits except that bits 14 and 15 are not used.

13.5.2.7 PATTERNS.

This table is set up by ADSIRP to contain the appropriate standard patterns for the disk type, and the patterns specified by the user.

```
   *-------------+-------------*
   |   PATTERN1 FROM DITDAT    |
   *-------------+-------------*
   |   PATTERN2 FROM DITDAT    |
   *-------------+-------------*
   | PTRN3 FROM DITDAT OR USER |
   *-------------+-------------*
   | PTRN4 FROM DITDAT OR USER |
   *-------------+-------------*
```

reserve block

13.5.2.8 HEAD_ERROR_FLAG.

There is one byte entry for each head. When the analysis on a cylinder begins, each entry is zero.

If a track on a cylinder has an error, then HEAD_ERROR_FLAG (for that track) equals 1.

If a track is determined bad at any time in the analysis, then HEAD_ERROR_FLAG (for that track) equals >FF.

```
*--------------------*
| ZERO, ONE, OR >FF  |
*--------------------*
/                    /
/                    /
/                    /
*--------------------*
```

13.5.2.9 BAD_RDS_PATTERN and TOT_RDS_PATTERN.

There are two tables of this format, BAD_RDS_PATTERN and TOT_RDS_PATTERN. They are used to count bad reads and total reads for each format pattern and each read during the analysis of a cylinder.

```
 PTRN1          PTRN2          PTRN3          PTRN4
*------+-------*------+-------*------+-------*------+-------*
| COUNT        |  COUNT       |  COUNT       |  COUNT       | HD 0
*------+-------*------+-------*------+-------*------+-------*
/              /              /              /              /
/              /              /              /              /
/              /              /              /              /
*------+-------*------+-------*------+-------*------+-------*
|              |              |              |              | HD 31
*------+-------*------+-------*------+-------*------+-------*
```

13.5.2.10 REC_DEFECT_LENGTH.

This is a byte table with one entry for each head. It is initialized to 2 for disks without SA-Assist and is never changed. For disks with SA-Assist, it contains the largest defect length returned by the SA-Assist command when an error is detected. If the track is determined bad, the information stored in this table is used to update the bad track table.

```
*-----------------------*
| LARGEST DEFECT LENGTH |
*-----------------------*
/                       /
/                       /
/                       /
*-----------------------*
```

13.5.2.11 IDS Global Data.

NUM_RD_TYPES	Count of the number of read types types used. Initialized in IDSIRP.
NUM_PATTERNS	Set based on the level of testing. Initialized in IDSIRP.
WRTFMTS_PER_PATTERN	Set based on the level of testing.

	Initialized in IDSIRP.
SA_ASSIST	Flag yes or no. Initialized in IDSIRP.
TOTAL_HEADS	Initialized in IDSIRP.
TOTAL_CYLINDERS	Initialized in IDSIRP.
GRAPH_FLAG	Flag yes or no. Initialized in IDSIRP.
@IDSPRM	Passed in register 3. Initialized in IDSIRP.
@BADTAB	From parameter block. Initialized in IDSIRP.
TOT_RDS_WRTFMT	Count of the total reads after each write format.
BAD_RDS_WRTFMT	Count of bad reads after each write format.
PTRN_INDX	Index into pattern lists. Current pattern to analyze.
ALLOWABLE_TRACK_ PERCENTAGE	10 or 30, depending on the mark marginal tracks flag in IDSPRM. This variable is used to verify that the number of errors detected during analysis of a track does not exceed 10% (if mark marginal tracks is yes) or 30%. Initialized in IDSIRP.
CURGRAPH	Contains position of last graph update. Must be initialized in INITIALIZE_ GRAPH_DISPLAY.
SECTOR_BUFFER	256 bytes overlap TOT_RDS_PATTERN because they are never used at the same time.
REPETITIONS	This word is set to 5 whenever an error is first encountered during the analysis. Reset to 1 before beginning the analysis for every cylinder.
REFORMAT_FLAG	Set to ones to cause simulation of transfer inhibit. If a disk does not support transfer inhibit, the disk will be formatted at one record per track. During an analysis, reads will use an input count of 2. This causes the entire track to be checked, but no huge buffers are needed. Initialized in IDSIRP.

SECTION 14

DATA STRUCTURE PICTURES

14.1 OVERVIEW

This section includes detailed pictures of special-purpose data structures used by SCI and the utilities. These templates are in the DSC.TEMPLATE.ATABLE directory, which includes templates for structures used throughout the operating system as well as templates for special purposes in a single subsystem or utility.

Details of operating system data structures appear in the Data Structures Pictures section of the DNOS System Design Document.

The template pictures include descriptions of various fields of data structures -- their locations, meanings of flags, and special comments. The following features are found in one or more of the structure pictures:

* Header showing the structure name, location in the system, and abbreviation for the name

* Comments describing the use of the structure

* Hexadecimal starting location (or offset relative to the beginning of the structure) for each word of the structure

* Label for each field, chosen from three types:

  - Blank if no label

  - Label of the form FILLxy, if the label is generated by software

  - Label of six or fewer characters

* Size of field indicated by space allocated in structure picture

* Comment to right of field, describing that field

* List of flag definitions for each flag field in the structure:

  - Flag name

- Diagram showing position of flag, initial position being 0. The flag is always defined as an assembly language equate for the first bit position shown with an X in the diagram.
- Description of flag
- Optional lines of extended explanations of flag settings

* List of equated labels for fields in the structure:
  - Label being equated
  - Argument of the equate
  - Value of the equate, or location of the argument
  - Description of the label being equated

Table 14-1 lists the templates detailed in this section.

Table 14-1 Template Acronyms

Acronym	Meaning
ACC	Accounting record contents
CLR	Capabilities list file record
CNT	Class name table
FIR	File information record
SCA	System Communication Area
SDEDOR	Memory-Resident directory overhead record
SDEMD	Sorted directory file entries table
SDQ	Spooler device queue entry
SDT	Spooler device table entry
SPM	Spooler message format
UDR	User descriptor record

```
***********************************************************
*                                                         *
*  ACCOUNTING RECORD CONTENTS (ACC)            09/09/83   *
*                                                         *
*          LOCATION:  SYSTEM TABLE AREA OR DISK           *
***********************************************************
* THE ACC DESCRIBES THE FORMAT OF ENTRIES ON THE QUEUE FOR
* PROCESSING BY THE ACCOUNTING FORMATTING TASK (LGACCT).
* WITH THE EXCEPTION OF THE QUEUE LINK, THE ENTRIES ARE
* EXACTLY THE SAME WHEN ON DISK IN THE ACCOUNTING LOG FILE.
* EACH BLOCK TYPE HAS ITS OWN SET OF INFORMATION FOLLOWING
* A STANDARD HEADER.  THE EXCEPTION IS IPL (RECORD TYPE 6),
* WHICH USES ONLY THE HEADER INFORMATION.

                              FIXED PART
      *----------+----------*
  >00 !       ACCLNK        !   QUEUE LINK
      +----------+----------+

                              FIELD DESCRIPTOR VARIANT
      *----------+----------*
  >02 ! ACCTYP   ! ACCLEN   !   RECORD TYPE
      +----------+----------+      LENGTH OF RECORD
  >04 !       ACCYRD        !   YEAR/DAY
      +----------+----------+
  >06 ! ACCHOU   ! ACCMIN   !   HOUR
      +----------+----------+      MINUTE
  >08 ! ACCSEC   ! ACCPRI   !   SECOND
      +----------+----------+      PRIORITY
  >0A !       ACCJID        !   JOB ID
      +----------+----------+

                              TYPE 1 - JOB INITIALIZATION
      *----------+----------*
  >0C ! ACCAID   !          !   ACCOUNT ID
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >1C ! ACCUID   !          !   USER ID
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >24 ! ACCJNM   !          !   JOB NAME
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+

                              TYPE 2 - TASK TERMINATION
      *----------+----------*
```

```
 >0C ! ACCTID  ! ACCTCD  !   TASK ID
     +---------+---------+      TASK TERM CODE
 >0E !      ACCCPU       !   TASK CPU TIME (CLOCK TICKS)
     +---------+---------+
 >10 !                   !
     +---------+---------+
 >12 !      ACCSVC       !   NUMBER SVC'S ISSUED
     +---------+---------+
 >14 !                   !
     +---------+---------+
 >16 !      ACCIOB       !   NUMBER I/O BYTES TRANFERED
     +---------+---------+
 >18 !                   !
     +---------+---------+
 >1A !      ACCMEM       !   MAX MEMORY ALLOCATED(BEETS)
     +---------+---------+
 >1C !      ACCWAL       !   WALL CLOCK EXECUTION TIME
     +---------+---------+
 >1E !                   !
     +---------+---------+
 >20 ! ACCIID  ! ACCSTN  !   INSTALLED TASK ID
     +---------+---------+      STATION ID
 >22 !      ACCATR       !   TASK ATTRIBUTES
     +---------+---------+
 >24 ! ACCTNM  !         !   TASK NAME
     +---------+---------+
     /         /         /
     /         /         /
     +---------+---------+

                       TYPE 3 - JOB TERMINATION
     *---------+---------*
 >0C !      ACCJUD       !   JCA AREA USED
     +---------+---------+
 >0E !      ACCJSZ       !   JCA TOTAL SIZE
     +---------+---------+
 >10 !      ACCJEX       !   JOB EXECUTION TIME
     +---------+---------+
 >12 !                   !
     +---------+---------+

                       TYPE 4 - DEVICE ENTRY
     *---------+---------*
 >0C ! ACCTPF  ! ACCDTP  !   DEVICE TYPE FLAGS
     +---------+---------+      DEVICE TYPE
 >0E ! ACCNAM  !         !   DEVICE NAME
     +---------+---------+
 >10 !         !         !
     +---------+---------+
 >12 !      ACCNRQ       !   NUMBER I/O REQUESTS
     +---------+---------+
 >14 !                   !
     +---------+---------+
>16 !       ACCTMU         !   RESERVED-TIME USED(MINUTES)
    +----------+----------+

                          TYPE 5 - USER ENTRY
    *----------+----------*
>0C !  ACCCHR  !          !   USER DATA
    +----------+----------+
    /          /          /
    /          /          /
    +----------+----------+

                          TYPE 7 - COMM ENTRY
    *----------+----------*
>0C !  ACCCOM  !          !   COMM DATA
    +----------+----------+
    /          /          /
    /          /          /
    +----------+----------+

                          SINGLE ENTITY VARIANT
    *----------+----------*
>02 !  ACCTXT  !          !
    +----------+----------+
    /          /          /
    /          /          /
    +----------+----------+

FLAGS FOR FIELD: ACCYRD    #04 - YEAR/DAY

  ACCYER = (XXXXXXX.........) - YEAR (7 BITS)
  ACCDAY = (.......XXXXXXXXX) - DAY (9 BITS)
```

EQUATES:

LABEL	EQUATE TO	VALUE	DESCRIPTION
ACCVNT	$	>02	
ACTJIT	1	>01	JOB INITIALIZATION
ACTTTM	2	>02	TASK TERMINATION
ACTJTM	3	>03	JOB TERMINATION
ACTDET	4	>04	DEVICE ENTRY
ACTUET	5	>05	USER ENTRY
ACTIPL	6	>06	IPL ENTRY
ACCOHD	$	>0C	END OF OVERHEAD
ACCJIZ	$	>2C	
ACCTTZ	$	>2C	
ACCJTZ	$	>14	
ACCDSZ	$	>18	
ACCUSZ	$	>52	
ACCISZ	$	>0C	
ACCCTZ	$	>52	

```
**********************************************************
*                                                        *
*      CAPABILITIES LIST FILE RECORD (CLR)      01/21/83 *
*                                                        *
*         LOCATION .S$CLF ON DISK                        *
*                                                        *
**********************************************************
* THE CLR IS USED BY TASKS WHICH ADD, DELETE, OR MODIFY
* USER IDS OR ACCESS GROUPS.  IT HAS 5 VARIANTS: FIR, AGR,
* UDR, UDO, AND VFY.  THE STRUCTURE AND PURPOSE OF EACH VARIANT
* IS DESCRIBED BELOW.
*
*
*  THIS PACKED RECORD IS USED FOR USER ID ENTRIES IN FIR
*
                               ** BEGINNING PACKED RECORD UID


      *----------+----------*
  >00 !  FIRID   !          !   USER ID
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >08 !        FIRRN        !   USER'S UDR RECORD NUMBER
      +----------+----------+
  >0A SIZE                 ** END OF PACKED RECORD

*
*  THIS PACKED RECORD IS USED FOR ACCESS GROUP ENTRIES IN
*  USER DESCRIPTOR RECORDS (UDR) AND USER DESCRIPTOR OVERFLOW
*  RECORDS (UDO).
*
                               ** BEGINNING PACKED RECORD AGE

                                  ACCESS GROUP ENTRY
      *----------+----------*
  >00 !        AGERN        !   ACCESS GROUP RECORD NUMBER
      +----------+----------+
  >02 !  AGEOFF  !  AGEFLG  !   OFFSET INTO ACCESS GROUP RECORD
      +----------+----------+      ACCESS GROUP ENTRY FLAGS
  >04 SIZE                 ** END OF PACKED RECORD
```

```
*
*  THIS PACKED RECORD IS USED FOR ACCESS GROUP NAMES IN
*  ACCESS GROUP RECORDS (AGR)
*
                                ** BEGINNING PACKED RECORD AGN


       *----------+----------*
  >00  !  AGNNAM  !          !    ACCESS GROUP NAME
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
  >08  !       AGNRSV        !    RESERVED
       +----------+----------+
  >0A  SIZE                 ** END OF PACKED RECORD

*
                                ** BEGINNING PACKED RECORD CLR

*
*
```

```
*
*         FILE INFORMATION RECORD (FIR)
*
*  THIS VARIANT IS USED TO STORE USER IDs.  IT CONTAINS A
*  FLAG WORD, A POINTER TO ANOTHER FIR, AND 5 UID ENTRIES.
*  EACH UID ENTRY CONTAINS A USER ID AND THE RECORD NUMBER
*  OF ITS USER DESCRIPTOR RECORD (UDR).
*
*


       *----------+----------*
   >00 !       FIRFIR        !   CONTINUATION RECORD NUMBER
       +----------+----------+
   >02 !       FIRRSV        !   FIR USED/AVAILABLE FLAG
       +----------+----------+
   >04 !       FIRENT        !   5 UID ENTRIES
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
   >0E !                     !
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
   >18 !                     !
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
   >22 !                     !
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
   >2C !                     !
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
*
*        ACCESS GROUP NAME RECORD (AGR)
*
*   THIS VARIANT IS USED TO STORE ACCESS GROUP NAMES.
*   IT CONTAINS A FLAG WORD, A POINTER TO THE NEXT AGR, AND
*   5 AGN ENTRIES.  EACH AGN ENTRY CONTAINS AN ACCESS GROUP
*   NAME AND A WORD OF UNUSED FLAGS.
*


        *----------+----------*
   >00  !        AGRAGR       !    CONTINUATION RECORD NUMBER
        +----------+----------+
   >02  !        AGRRSV       !    AGR USED/AVAILABLE FLAG
        +----------+----------+
   >04  !        AGRAGN       !    5 AGN ENTRIES
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
   >0E  !                     !
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
   >18  !                     !
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
   >22  !                     !
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
   >2C  !                     !
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
*
*        USER DESCRIPTOR RECORD (UDR)
*
*  THIS VARIANT CONTAINS INFORMATION ASSOCIATED WITH A USER ID.
*  THIS INFORMATION INCLUDES THE ENCRYPTED PASSCODE, DESCRIPTION,
*  AND UP TO 5 ACCESS GROUP ENTRIES.  EACH ACCESS GROUP ENTRY
*  CONTAINS A RECORD NUMBER OF AN ACCESS GROUP RECORD (AGR)
*  AND THE OFFSET INTO THE AGR FOR AN ACCESS GROUP NAME OF
*  WHICH THIS USER IS A MEMBER.
*


      *----------+----------*
  >00 !       UDRUDO        !   POINTER TO OVERFLOW
      +----------+----------+
  >02 !       UDRRSV        !   UDR USED/AVAILABLE FLAG
      +----------+----------+
  >04 !  UDRPWD  !          !   ENCRYPTED PASSCODE
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >0C !       UDRFLG        !   UDR FLAG WORD
      +----------+----------+
  >0E !  UDRDES  !          !   DESCRIPTION OF USER
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >22 !       UDRAGE        !   5 ACCESS GROUP ENTRIES (AGE)
      +----------+----------+
  >24 !          !          !
      +----------+----------+
  >26 !                     !
      +----------+----------+
  >28 !          !          !
      +----------+----------+
  >2A !                     !
      +----------+----------+
  >2C !          !          !
      +----------+----------+
  >2E !                     !
      +----------+----------+
  >30 !          !          !
      +----------+----------+
  >32 !                     !
      +----------+----------+
  >34 !          !          !
      +----------+----------+
*
*        USER DESCRIPTOR OVERFLOW RECORD (UDO)
*
*  THIS VARIANT IS USED ONLY USED IN THE CASE THAT A USER IS
*  A MEMBER OF MORE ACCESS GROUPS THAN WILL FIT IN HIS UDR.
*  IT CONTAINS UP TO 12 ACCESS GROUP ENTRIES.
*


      *----------+----------*
  >00 !        UDOUDO       !   POINTER TO NEXT UDO
      +----------+----------+
  >02 !        UDORSV       !   UDO USED/AVAILABLE FLAG
      +----------+----------+
  >04 !        UDOFIL       !   NOT USED
      +----------+----------+
  >06 !        UDOAGE       !   12 ACCESS GROUP ENTRIES (AGE)
      +----------+----------+
  >08 !          !          !
      +----------+----------+
  >0A !                     !
      +----------+----------+
  >0C !          !          !
      +----------+----------+
  >0E !                     !
      +----------+----------+
  >10 !          !          !
      +----------+----------+
  >12 !                     !
      +----------+----------+
  >14 !          !          !
      +----------+----------+
  >16 !                     !
      +----------+----------+
  >18 !          !          !
      +----------+----------+
  >1A !                     !
      +----------+----------+
  >1C !          !          !
      +----------+----------+
  >1E !                     !
      +----------+----------+
  >20 !          !          !
      +----------+----------+
  >22 !                     !
      +----------+----------+
  >24 !          !          !
      +----------+----------+
  >26 !                     !
      +----------+----------+
  >28 !          !          !
      +----------+----------+
  >2A !                     !
```

```
     +----------+----------+
>2C  !          !          !
     +----------+----------+
>2E  !                     !
     +----------+----------+
>30  !          !          !
     +----------+----------+
>32  !                     !
     +----------+----------+
>34  !          !          !
     +----------+----------+
*
*        VERIFICATION RECORD (VFY)
*
*  THIS VARIANT IS USED BY THE SYSTEM RESTART TASK TO VERIFY
*  THE EXISTENCE OF .S$CLF.  IT IS ALSO USED BY TASKS WHICH
*  CREATE AND MODIFY ACCESS GROUPS BECAUSE IT CONTAINS A
*  POINTER TO THE FIRST ACCESS GROUP RECORD.
*


     *----------+----------*
>00  !  VFYNAM  !          !   NAME OF S$CLF
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
>08  !       VFYBLK        !   POINTER TO FIRST AGRBLK
     +----------+----------+
>0A  !  VFYFIL  !          !   NOT USED, INITIALIZED TO BLANKS
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
>36 SIZE                ** END OF PACKED RECORD


FLAGS FOR FIELD: AGEFLG    #03 - ACCESS GROUP ENTRY FLAGS

  AGELDR = (X...............) - TRUE=USER IS LEADER OF ACCESS GROUP
  AGEFCG = (.X..............) - TRUE=FILE CREATION ACCESS GROUP


FLAGS FOR FIELD: FIRRSV    #02 - FIR USED/AVAILABLE FLAG

  FIRFRE = (X...............) - TRUE=AVAILABLE RECORD


FLAGS FOR FIELD: AGRRSV    #02 - AGR USED/AVAILABLE FLAG

  AGRFRE = (X...............) - TRUE=AVAILABLE RECORD


FLAGS FOR FIELD: UDRRSV    #02 - UDR USED/AVAILABLE FLAG

  UDRFRE = (X...............) - TRUE=AVAILABLE RECORD


FLAGS FOR FIELD: UDRFLG    #0C - UDR FLAG WORD

  UDRPVL = (XXXXX...........) - USER PRIVELEDGE LEVEL
  UDRAGC = (.....XXXXXXXXXXX) - ACCESS GROUP COUNT
```

```
FLAGS FOR FIELD: UDORSV    #02 - UDO USED/AVAILABLE FLAG

  UDOFRE = (X...............) - TRUE=AVAILABLE ENTRY
```

EQUATES:

LABEL	EQUATE TO	VALUE	DESCRIPTION
FIR	$	>00	
AGR	$	>00	
UDR	$	>00	
UDO	$	>00	
VFY	$	>00	
FIRSIZ	$	>36	
AGRSIZ	$	>36	
UDRSIZ	$	>36	
UDOSIZ	$	>36	
VFYSIZ	$	>36	

```
***********************************************************
*                                                         *
*      CLASS NAME TABLE ENTRY  (CNT)             09/09/83 *
*                                                         *
*           LOCATION: .S$SDTQUE.(SYSNAME) AND             *
*                          SPOOLER TASK COMMON (SPCOMN)   *
***********************************************************
* THE CNT IS USED BY THE SPOOLER TO SAVE CLASS NAME
* INFORMATION.
                            ** BEGINNING PACKED RECORD CNT


       *----------+----------*
 >00 !  CNTFLG  !  CNTPRI  !
       +----------+----------+      QUEUE ELEMENT PRIORITY
 >02 !        CNTDEV         !   COUNT OF DEVICES USING CLASS NAME
       +----------+----------+
 >04 !        CNTRN          !   QUEUE ELEMENT RECORD NUMBER
       +----------+----------+
 >06 !  CNTOFF  !  CNTRES  !   QUEUE ELEMENT RECORD OFFSET
       +----------+----------+      *** RESERVED ***
 >08 !  CNTNAM  !            !   CLASS NAME
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
 >10 SIZE                  ** END OF PACKED RECORD


 FLAGS FOR FIELD: CNTFLG    #00 -

   CNFDEL = (X...............) - TRUE=DELETED ENTRY
          = (.X..............) -
   CNFHLT = (..X.............) - TRUE=CLASS IS HALTED

 EQUATES:

    LABEL    EQUATE TO    VALUE    DESCRIPTION
    -----    ---------    -----    ------------------------------
    CNTNUM   48             >30      NUMBER OF CNT ENTRIES
```

```
**********************************************************
*                                                        *
*  FILE INFORMATION RECORD     (FIR)        11/24/82     *
*                                                        *
*          LOCATION: DISK                                *
**********************************************************
* THE FIR IS USED BY THE TASKS WHICH ASSIGN, MODIFY, LIST,
* AND DELETE USER IDS.  IT IS A VARIANT OF THE CAPABILITIES
* LIST FILE RECORD (CLR).  FOR DETAILS SEE CLR.
```

```
******************************************************************
* SYSTEMS COMMUNICATION AREA (SCA)    1/09/80
*
* TERMINAL ENTRY DEFINITION ON FILE .S$SCA
*
******************************************************************


      *----------+----------*
  >00 !  SCADN   !          !   SCA DEVICE NUMBER
      +----------+----------+
  >02 !          !          !
      +----------+----------+
  >04 !  FILL00  !  SCAUID  !
      +----------+----------+      DEFAULT USER ID
      /          /          /
      /          /          /
      +----------+----------+
  >0C !          !  FILL01  !
      +----------+----------+
  >0E !  SCACCD  !          !   DEFAULT ACCOUNT NUMBER
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >1E !  FILL02  !  SCAPSD  !
      +----------+----------+      DEFAULT PASSCODE
      /          /          /
      /          /          /
      +----------+----------+
  >26 !          !  FILL03  !
      +----------+----------+
  >28 !  SCAJND  !          !   DEFAULT JOB NAME
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >30 !  FILL04  !  FILL05  !
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >48 !  SCALG   !  SCAMD   !   LOGIN REQUIRED
      +----------+----------+      VDT MODE
  >4A !  SCAJN   !  SCARC   !   DON'T SOLICIT JOB NAME
      +----------+----------+      RECONNECT DISABLED
  >4C !  SCAAC   !  SCASL   !   SOLICIT ACCOUNT NUMBER
      +----------+----------+      SOLICIT NAME MANAGER FILES
  >4E !  SCAOFF  !  SCADFM  !   TERMINAL OFF
      +----------+----------+      VDT MODE DEFAULT
```

EQUATES:

LABEL	EQUATE TO	VALUE	DESCRIPTION
SCADFS	$-SCADN	>31	
SCAFLG	8	>08	NUMBER OF FLAGS
SCASIZ	80	>50	

```
********************************************************************
*  DIRECTORY OVERHEAD RECORD  (SDEDOR)
*
*            MEMORY RESIDENT TABLE FORM OF DOR  09/25/79
********************************************************************


     *----------+----------*
>00  !       SDONRC        !    # RECORDS IN DIRECTORY
     +----------+----------+
>02  !       SDONFL        !    # FILES CURRENTLY IN DIRECTORY
     +----------+----------+
>04  !       SDONAR        !    # OF AVAILABLE RECORDS
     +----------+----------+
>06  !       SDOTFC        !    NUMBER OF TEMPORARY FILES
     +----------+----------+
>08  !  SDODNM  !          !    DIRECTORY FILE NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
>10  !       SDOLVL        !    LEVEL # OF DIRECTORY
     +----------+----------+
>12  !  SDOPNM  !          !    PARENT'S NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
>1A  !       SDORDH        !    DIRECTORY ENTRY LIST HEADER
     +----------+----------+
>1C  !       SDORFH        !    FILE ENTRY LIST HEADER
     +----------+----------+
>1E  !       SDORCH        !    CHANNEL ENTRY LIST HEADER
     +----------+----------+
>20  !       SDOFLG        !    MODIFIED FORMAT FLAG
     +----------+----------+       0=NORMAL FORMAT

EQUATES:

  LABEL   EQUATE TO   VALUE   DESCRIPTION
  -----   ---------   -----   ------------------------------
  SDOSIZ  $             >1A
```

```
******************************************************
*  SORTED DIRECTORY FILE ENTRIES (SDEMD)
*      FOR MAP DISC AND LIST DIRECTORY
*           09/25/79
******************************************************


      *----------+----------*
  >00 !  SDEFNM  !          !   FILE NAME
      +----------+----------+
      /          /          /
      /          /          /
      +----------+----------+
  >08 !       SDEREC        !   FDR RECORD NUMBER
      +----------+----------+
  >0A !       SDEFLG        !   FLAGS
      +----------+----------+
  >0C !       SDEPRS        !   PHYSICAL RECORD SIZE
      +----------+----------+
  >0E !       SDELRS        !   LOGICAL RECORD SIZE
      +----------+----------+
  >10 !       SDEPAS        !   PRIMARY ALLOCATION SIZE
      +----------+----------+
  >12 !       SDESAS        !   SECONDARY ALLOCATION SIZE
      +----------+----------+
  >14 !       SDESAA        !   OFFSET OF SCONDARY TABLE
      +----------+----------+
  >16 !       SDERFA        !   RECORD # OF FIRST ALIAS
      +----------+----------+
  >18 !  SDEEOM  !          !   END OF MEDIUM RECORD #
      +----------+----------+
  >1A !          !          !
      +----------+----------+
  >1C !  SDEBKM  !          !   END OF MEDIUM BLOCK #
      +----------+----------+
  >1E !          !          !
      +----------+----------+
  >20 !       SDEOFM        !   END OF MEDIUM OFFSET
      +----------+----------+
  >22 !       SDETNB        !   TOTAL # OF BLOCKS
      +----------+----------+
  >24 !       SDEKDR        !   KEY DESCRIPTIONS RECORD #
      +----------+----------+
  >26 !  SDEUD   !          !   LAST UPDATE DATE
      +----------+----------+
  >28 !          !          !
      +----------+----------+
  >2A !          !          !
      +----------+----------+
  >2C !  SDECD   !          !   CREATION DATE
      +----------+----------+
  >2E !          !          !
```

```
     +----------+----------+
>30  !          !          !
     +----------+----------+
>32  !  SDEAPB  !  SDEBPA  !   ADU'S PER BLOCK
     +----------+----------+      BLOCK'S PER ADU
>34  !        SDELNK       !   LINK TO NEXT ENTRY
     +----------+----------+
>36  !        SDEALO       !   TOTAL FILE ALLOCATION IN ADUS
     +----------+----------+
>38  !        SDEUSE       !   # OF USED SECONDARY ENTRIES
     +----------+----------+


     *----------+----------*
>00  !  MODFNM  !          !   FILE NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
>08  !        MODREC       !   RECORD NUMBER OF FILE
     +----------+----------+
>0A  !        MODRAF       !   POINTER TO ACTUAL FILE NAME
     +----------+----------+
>0C  !        MODLNK       !   LINK TO NEXT MODIFIED ENTRY
     +----------+----------+
```

EQUATES:

LABEL	EQUATE TO	VALUE	DESCRIPTION
SDEIID	SDEPRS	>0C	OWNER TASK INSTALLED ID
SDECFL	SDEIID+1	>0D	CHANNEL FLAGS
SDETF	SDELRS	>0E	CHANNEL RESOURCE FLAGS
SDEMXL	SDEPAS	>10	CHANNEL MAX MSG LENGTH
SDERAF	$	>12	RECORD # OF ACTUAL FILE
SDERNA	$	>16	RECORD # OF NEXT ALIAS
SDESIZ	$	>3A	SIZE IN BYTES OF FDR
MODFLG	$	>08	FILE TYPE FLAGS
MODFSZ	$	>0A	FILE SIZE IN ADUS
MODSIZ	$	>0E	SIZE IN BYTES

```
*****************************************************
*       SDQ -- SPOOLER DEVICE QUEUE ENTRY           *
*            04/07/82                               *
*****************************************************
*
*
                          ** BEGINNING PACKED RECORD SDQ


        *----------+----------*
    >00 !        SDQFLG       !   STATUS FLAGS
        +----------+----------+
    >02 !        SDQRN        !   NEXT ENTRY RECORD NUMBER
        +----------+----------+
    >04 !  SDQOFF  !  SDQCOF  !   NEXT ELEMENT RECORD OFFSET
        +----------+----------+      NEXT FILENAME RECORD OFFSET
    >06 !        SDQCRN       !   NEXT FILENAME RECORD
        +----------+----------+


        *----------+----------*
    >08 !  SDQULN  !          !   USER LOGICAL NAME
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
    >10 !  SDQFRM  !          !   REQUESTED FORM
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
    >18 !  SDQUID  !          !   USER ID
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
    >20 !        SDQJID       !   JOB ID
        +----------+----------+
    >22 !  SDQJOB  !          !   JOB NAME
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+


        *----------+----------*
    >2A !  SDQCLN  !          !   CLASS NAME
        +----------+----------+
        /          /          /
        /          /          /
        +----------+----------+
    >32 !  SDQCOP  !  SDQLPP  !   NUMBER OF COPIES
```

```
         +----------+----------+       LINES PER PAGE
    >34  !  SDQJPR  !  SDQNPR  !    JOB PRIORITY
         +----------+----------+       NEXT ENTRY PRIORITY
    >36  !  SDQSID  !          !    ASSIGNED SPOOLER ID
         +----------+----------+
    >38  !          !          !
         +----------+----------+
    >3A  !          !          !
         +----------+----------+
    >3C  !  SDQPNL  !          !    FILE PATHNAME
         +----------+----------+
         /          /          /
         /          /          /
         +----------+----------+


         *----------+----------*
    >2A  !  SDQDVN  !          !    DEVICE NAME
         +----------+----------+
         /          /          /
         /          /          /
         +----------+----------+


         *----------+----------*
    >08  !       SDQFNM        !    OF FILENAMES IN RECORD
         +----------+----------+
    >0A  !  SDQNAM  !          !    AREA RESERVED FOR FILE PATHNAMES
         +----------+----------+
         /          /          /
         /          /          /
         +----------+----------+
    >72  SIZE                ** END OF PACKED RECORD


    FLAGS FOR FIELD: SDQFLG    #00 - STATUS FLAGS

      SQFUSE = (X...............) - TRUE=DELETED ENTRY
      SQFCON = (.X..............) - TRUE=CONCATENATED FILE
      SQFANS = (..X.............) - TRUE=ANSI FILE
      SQFBNR = (...X............) - TRUE=NO BANNER SHEET PROVIDED
      SQFDEV = (....X...........) - TRUE=QUEUED FOR DEVICE
      SQFDAP = (.....X..........) - TRUE=DELETE AFTER PRINT
      SQFCFR = (......X.........) - TRUE=CONCAT FILENAME RECORD
      SQFDAL = (.......X........) - TRUE=DELETE ALWAYS (EVEN ON
*                                   KILL OUTPUT)


    EQUATES:

       LABEL   EQUATE TO   VALUE   DESCRIPTION
       -----   ---------   -----   ------------------------------
```

```
SDQNUM    6              >06    NUMBER OF SDQ ENTRIES PER RECORD
SDQVR1    $              >08
SDQVR2    $              >2A
```

```
************************************************************
*                                                          *
*      SPOOLER DEVICE TABLE ENTRY  (SDT)        09/09/83   *
*                                                          *
*           LOCATION: .S$SDTQUE.(SYSNAME) AND              *
*                      SPOOLER TASK COMMON (SPCOMN)        *
*                                                          *
************************************************************
* THE SDT IS USED BY THE SPOOLER TO SAVE DEVICE INFORMATION.
* ALL OF THE SPOOLER DEVICES ARE STORED ON DISK IN FILE
* .S$SDTQUE.(SYSNAME) WHERE (SYSNAME) IS THE NAME OF THE
* OS KERNAL. SPOOL DEVICE INFORMATION IN THE SDT INCLUDES
* STATUS FLAGS, CLASS NAME INFORMATION, FORM INFORMATION
* QUEUE INFORMATION AND PAGE INFORMATION.
*
                            ** BEGINNING PACKED RECORD DC


     *----------+----------*
 >00 !        SDTCN        !
     +----------+----------+
 >02 !        SDTCNR       !
     +----------+----------+
 >04 SIZE                   ** END OF PACKED RECORD

*
                            ** BEGINNING PACKED RECORD SDT


     *----------+----------*
 >00 !  SDTFLG  !  SDTLUN  !   STATUS FLAGS
     +----------+----------+      ASSIGNED JOB-LOCAL LUNO
 >02 !  SDTDNM  !          !   DEVICE NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
 >0A !  SDTAPR  !  SDTQPR  !   ACTIVE REQUEST PRIORITY
     +----------+----------+      QUEUED REQUEST PRIORITY
 >0C !        SDTARN       !   ACTIVE REQUEST RECORD
     +----------+----------+
 >0E !        SDTQRN       !   QUEUED REQUEST RECORD
     +----------+----------+
 >10 !  SDTAOF  !  SDTQOF  !   ACTIVE REQUEST RECORD OFFSET
     +----------+----------+      QUEUED REQUEST RECORD OFFSET
 >12 !  SDTAID  !          !   ACTIVE REQUEST SPOOL ID
     +----------+----------+
 >14 !          !          !
     +----------+----------+
 >16 !          !          !
     +----------+----------+
 >18 !        SDTCLS       !   CLASS NAME INDEXES
```

```
       +----------+----------+       MUST BE A PACKED ARRAY(1..6) OF DCL
  >1A  !          !          !
       +----------+----------+
  >1C  !                     !
       +----------+----------+
  >1E  !          !          !
       +----------+----------+
  >20  !                     !
       +----------+----------+
  >22  !          !          !
       +----------+----------+
  >24  !                     !
       +----------+----------+
  >26  !          !          !
       +----------+----------+
  >28  !                     !
       +----------+----------+
  >2A  !          !          !
       +----------+----------+
  >2C  !                     !
       +----------+----------+
  >2E  !          !          !
       +----------+----------+
  >30  !  SDTDTF  !  SDTTYP  !   DEVICE TYPE FLAGS
       +----------+----------+      DEVICE TYPE VALUE
  >32  !       SDTPAG        !   PAGES TO FORWARD/REVERSE
       +----------+----------+      NOTE: SDTPAG SIGN BIT 0 => FORWARD
  >34  !  SDTFRM  !          !   FORM MOUNTED ON DEVICE
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
  >3C  SIZE                ** END OF PACKED RECORD


  FLAGS FOR FIELD: SDTFLG    #00 - STATUS FLAGS

    SDFDEL = (X...............) - TRUE=DELETED ENTRY
    SDFAVB = (.X..............) - TRUE=NOT AVAILABLE TO SPOOLER
    SDFHLT = (..X.............) - TRUE=DEVICE HALTED
    SDFBSY = (...X............) - TRUE=DEVICE BUSY
    SDFFRM = (....X...........) - TRUE=DEVICE DOES NOT USE FORMS
    SDFKIL = (.....X..........) - TRUE=KILL OUTPUT REQUEST
*                               [WRITER SENDS 'I AM DONE']
    SDFTRM = (......X.........) - TRUE=WRITER TERMINATE
*                               [NO WRITER MESSAGE SENT]
    SDFSHR = (.......X........) - TRUE=REMOTE OR SHARED DEVICE


  EQUATES:

     LABEL   EQUATE TO   VALUE   DESCRIPTION
```

```
-----   ---------   -----   ------------------------------
SDTNUM  12            >0C     NUMBER OF SDT ENTRIES PER RECORD
```

```
************************************************************
*                                                          *
*  SPOOLER MESSAGE FORMAT (SPM)                  09/09/83  *
*                                                          *
*           LOCATION: .S$DSTCHN CHANNEL COMMUNICATION TO   *
*                     SPOOLER                              *
************************************************************
* THE SPM TEMPLATE IS USED TO DECODE INFORMATION PASSED TO
* THE SPOOLER TASK FROM THE PF TASK.
                        ** BEGINNING PACKED RECORD SPM


     *----------+----------*
 >00 !  SPMOPC  !  SPMERC  !   DST MESSAGE OP CODE
     +----------+----------+      DST RETURNED ERROR CODE
 >02 !  SPMUSR  !          !   USER ID
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
 >0A !  SPMJOB  !          !   JOB NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
 >12 !       SPMJID        !   JOB ID
     +----------+----------+
 >14 !       SPMFLG        !   STATUS FLAGS
     +----------+----------+

                           DEVICE NAME VARIANT
     *----------+----------*
 >16 !  SPMDVN  !          !   OUTPUT DEVICE NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+

                           CLASS NAME VARIANT
     *----------+----------*
 >16 !  SPMCLN  !          !   OUTPUT CLASS NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
 >1E !  SPMULN  !          !   USER LOGICAL NAME
     +----------+----------+
     /          /          /
     /          /          /
     +----------+----------+
 >26 !  SPMSLN  !          !   DST LOGICAL NAME (SPOOLER ID)
     +----------+----------+
   >28 !          !          !
       +----------+----------+
   >2A !          !          !
       +----------+----------+
   >2C !  SPMFRM  !          !   DESIRED FORM
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+
   >34 !  SPMCPY  !  SPMLPP  !   NUMBER OF COPIES
       +----------+----------+      LINES PER PAGE
   >36 !  SPMJPR  !  FILL01  !   JOB PRIORITY
       +----------+----------+      RESERVED
   >38 !       SPMPAG        !   FORWARD/REVERSE PAGE COUNT
       +----------+----------+

                            PATHNAME VARIANT
       *----------+----------*
   >3A !  SPMPTH  !          !   PATHNAME(S)
       +----------+----------+
       /          /          /
       /          /          /
       +----------+----------+

                            DEVICE USE VARIANT
       *----------+----------*
   >3A !       SPMIOC        !   DEVICE I/O COUNT
       +----------+----------+
   >3C !                     !
       +----------+----------+
   >3E !       SPMTIM        !   TIME DEVICE WAS USED
       +----------+----------+
>013A SIZE                ** END OF PACKED RECORD


   FLAGS FOR FIELD: SPMFLG    #14 - STATUS FLAGS

     SPFUSE = (X...............) - TRUE=DELETED ENTRY
     SPFAVL = (.X..............) - TRUE=NOT AVAILABLE TO SPOOLER
     SPFPGD = (..X.............) - TRUE=REVERSE PAGING
     SPFCON = (...X............) - TRUE=CONCATENATED FILE
     SPFSOP = (....X...........) - TRUE=SYSTEM OPERATOR
     SPFANS = (.....X..........) - TRUE=ANSI FORMAT
     SPFBNR = (......X.........) - TRUE=NO BANNER SHEET DESIRED
     SPFDAP = (.......X........) - TRUE=DELETE AFTER PRINT
     SPFIMM = (........X.......) - TRUE=HALT IMMEDIATELY
*                                 INSTEAD OF HALT AT EOF
     SPFABE = (.........X......) - TRUE=LPWRITER TASK ABENDED
     SPFDVE = (..........X.....) - TRUE=DEVICE ERROR OCCURRED
     SPFPFE = (...........X....) - TRUE=PRINT FILE ERROR
     SPFSHR = (............X...) - TRUE=REMOTE OR SHARED DEVICE
     SPFDAL = (.............X..) - TRUE=DELETE ALWAYS (EVEN IF
*                                   KILL OUTPUT DONE)
*
```

EQUATES:

LABEL	EQUATE TO	VALUE	DESCRIPTION
NVRNT	SPMFLG+2	>16	
PNVRNT	SPMJPR+4	>3A	

```
**********************************************************
*                                                        *
*  USER DESCRIPTOR RECORD      (UDR)        11/24/82     *
*                                                        *
*          LOCATION: DISK                                *
**********************************************************
* THE UDR DESCRIBES THE DISK STRUCTURES THAT REPRESENTS A
* GIVEN USER OF THE SYSTEM.  IT INCLUDES LOGON INFORMATION
* AND SECURITY INFORMATION.  IT IS A VARIANT OF THE CAPABILITIES
* LIST FILE RECORD (CLR).  FOR DETAILS SEE CLR.
```

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	ALT	ALT	ALT	None
Attention[2]	■	SHIFT PREV FORM NEXT	■	■	CTRL S
Back Tab	None	SHIFT TAB	SHIFT ⇧ TAB	None	CTRL T
Command[2]	■	PREV FORM NEXT	CMD	■	CTRL X
Control	CONTROL	CTRL	CTRL	CTRL	CTRL
Delete Character	DEL CHAR	LINE DEL CHAR	DEL CHAR	DEL CHAR	None
Enter	■	SEND	ENTER	ENTER	CTRL Y
Erase Field	ERASE FIELD	EOS ERASE EOF	ERASE FIELD	ERASE FIELD	CTRL] [

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	ERASE INPUT	ALL ERASE INPUT	ERASE INPUT	ERASE INPUT	CTRL N
Exit	ESC	PREV PAGE NEXT	SHIFT ⇧ ESC	SHIFT ESC	ESC
Forward Tab	SHIFT TAB SKIP	⇤ TAB ⇥	⇤ TAB ⇥	SHIFT TAB SKIP	CTRL I
F1	F1	F1	F1	F1	CTRL A
F2	F2	F2	F2	F2	CTRL B
F3	F3	F3	F3	F3	CTRL C
F4	F4	F4	F4	F4	CTRL D

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	F5	F5	F5	F5	CTRL E
F6	F6	F6	F6	F6	CTRL F
F7	F7	F7	F7	F7	CTRL V
F8	F8	F8	F8	F8	CTRL W
F9	CONTROL !1	F9	F9	SHIFT F1	CTRL][
F10	CONTROL @2	F10	F10	SHIFT F2	CTRL Z

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	CONTROL $ 4	F11	F11	SHIFT F3	CTRL | \
F12	CONTROL % 5	F12	F12	SHIFT F4	CTRL } {
F13	CONTROL ∧ 6	SHIFT F1	SHIFT ⇧ F1	SHIFT F5	CTRL + =
F14	CONTROL & 7	SHIFT F2	SHIFT ⇧ F2	SHIFT F6	CTRL _ -
Home	HOME	HOME	HOME	HOME	CTRL L
Initialize Input	▢	SHIFT LINE INS CHAR	▢	■	CTRL O

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	INS CHAR	LINE INS CHAR	INS CHAR	INS CHAR	None
Next Character	→ or SHIFT ⇄ CHAR	→	►	→	None
Next Field	SHIFT ⇄ FIELD	LINE FEED	SHIFT ⇧ ⇄ FIELD	SHIFT ⇄ FIELD	None
Next Line	↓	↓	▼	↓	CTRL J or LINE FEED
Previous Character	← or ⇄ CHAR	←	◄	←	None
Previous Field	⇄ FIELD	SHIFT ⇄ SKIP	⇄ FIELD	⇄ 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)

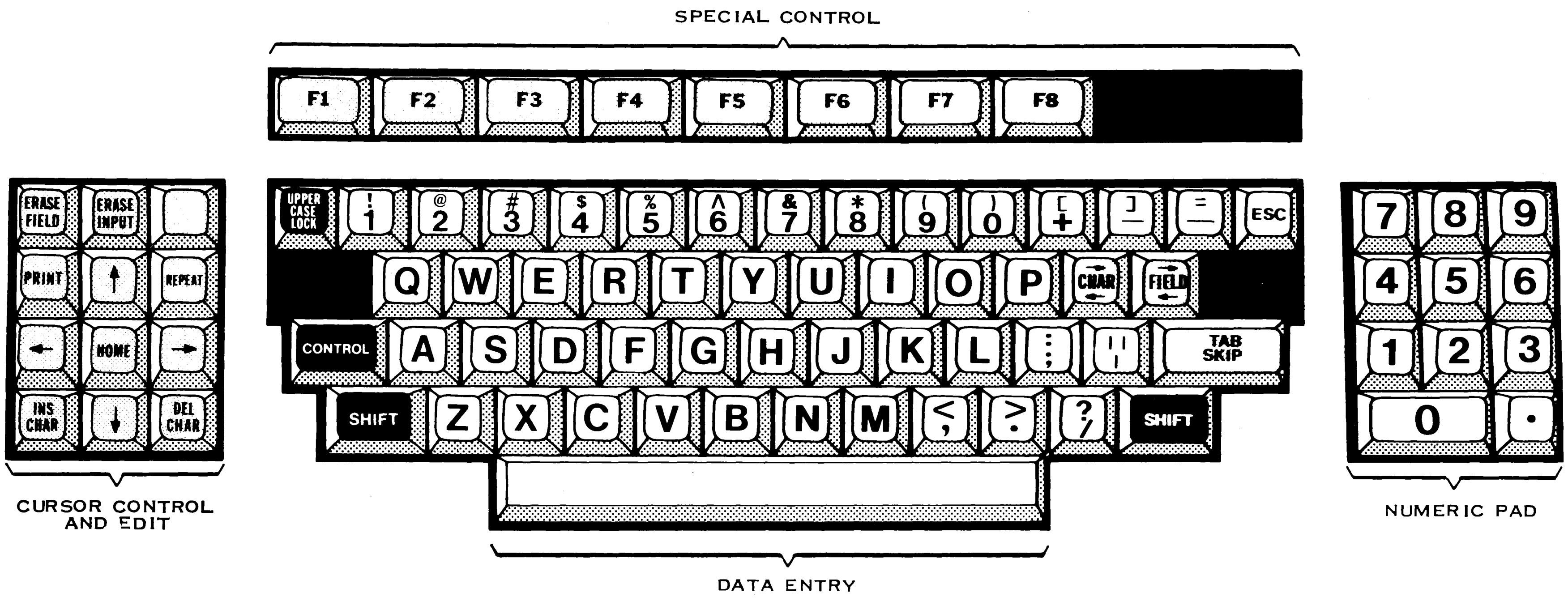
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Previous Line	↑	↑	▲	↑	CTRL U
Print	PRINT	PRINT	PRINT	PRINT	None
Repeat	REPEAT	See Note 3	See Note 3	See Note 3	None
Return	■	■	↵ RETURN	■	■
Shift	SHIFT	SHIFT	SHIFT ⇧	SHIFT	SHIFT
Skip	TAB SKIP	← SKIP →	SKIP	TAB SKIP	None
Uppercase Lock	UPPER CASE LOCK	UPPER CASE	CAPS LOCK	UPPER CASE LOCK	UPPER CASE

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.

[3]The keyboard is typamatic, and no repeat key is needed.

2284734 (7/14)



2284734 (9/14)

Figure A-1. 911 VDT Standard Keyboard Layout

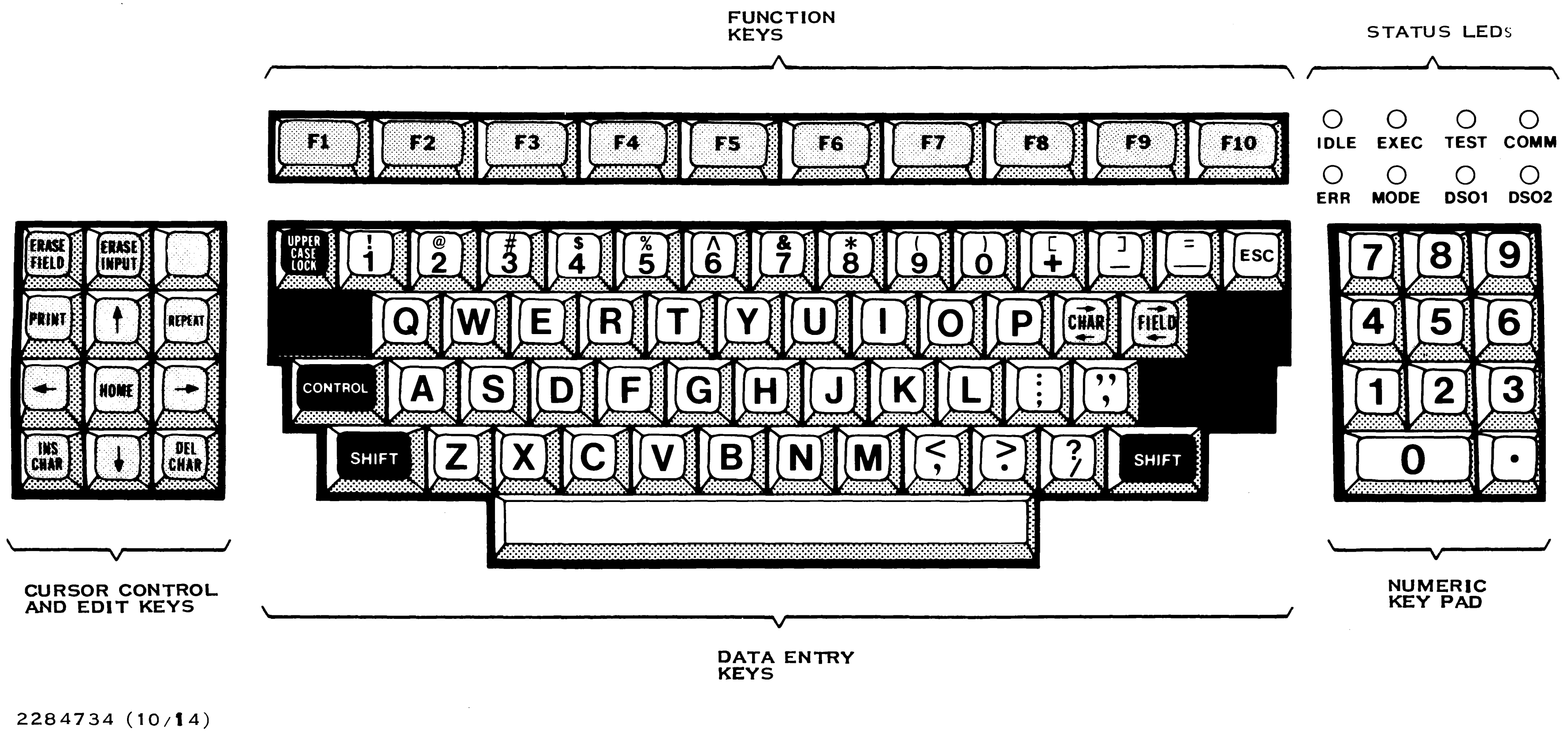
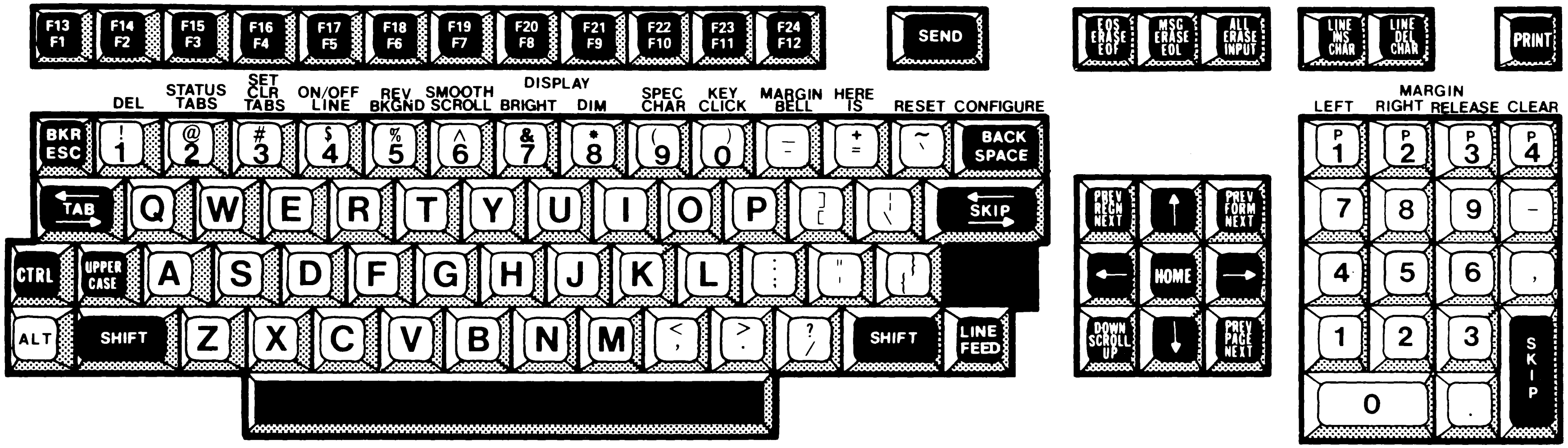
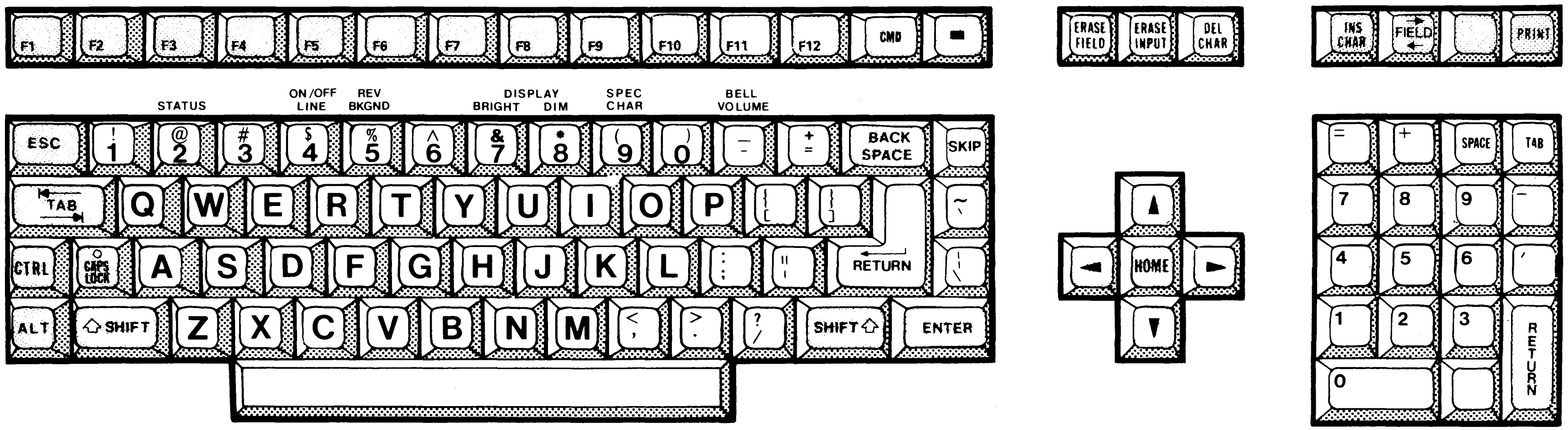


Figure A-2. 915 VDT Standard Keyboard Layout



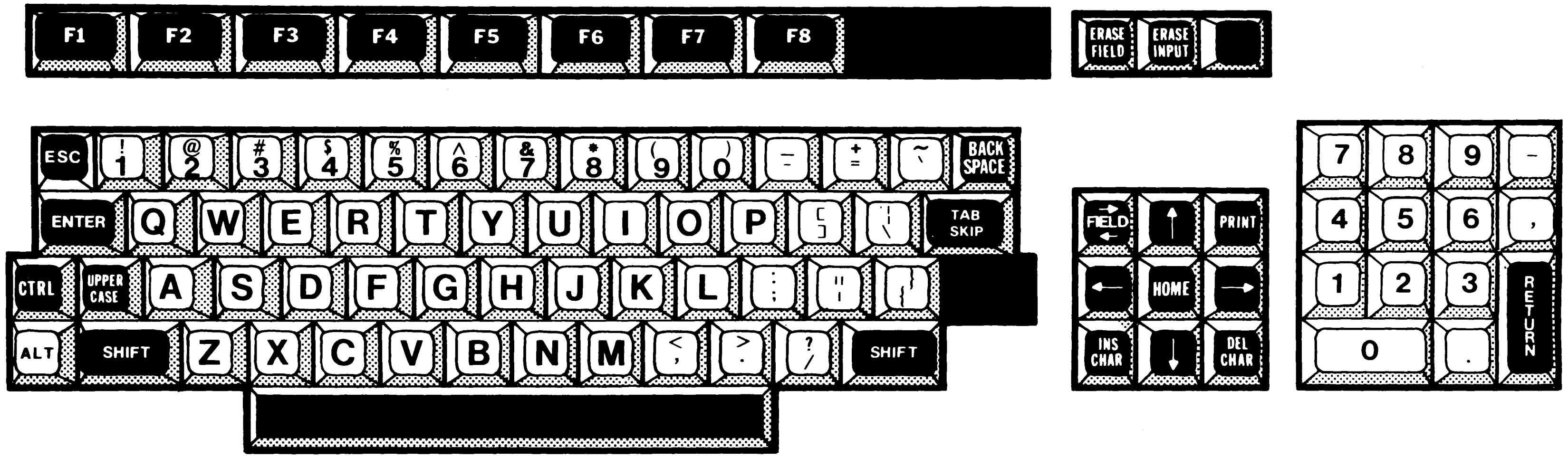
2284734 (11/14)

Figure A-3. 940 EVT Standard Keyboard Layout



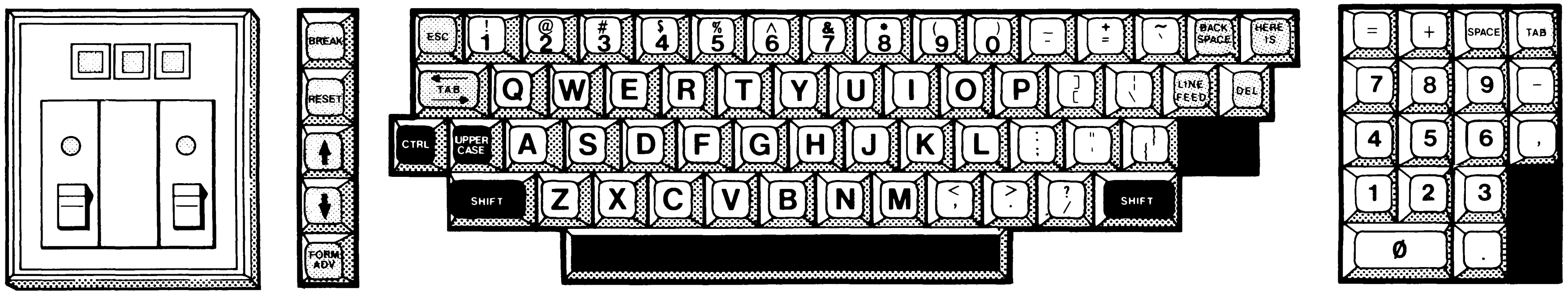
2284734 (12/14)

Figure A-4. 931 VDT Standard Keyboard Layout



2284734 (13/14)

Figure A-5. Business System Terminal Standard Keyboard Layout



2284734 (14/14)

Figure A-6. 820 KSR Standard Keyboard Layout

APPENDIX B

WRITING DSEG POSITION-INDEPENDENT CODE

B.1 OVERVIEW

The use of procedure segments that are shared by more than one task in a program file can result in significant savings of both disk space and memory required to execute the tasks concurrently. Special coding requirements must be met to allow this sharing. These requirements arise from the following conflicts:

* References within the procedure to variables defined in a DSEG are resolved when a task is linked. Different tasks may have DSEGs positioned (by the Link Editor) at different locations, relative to the beginning of each task segment.

* References within the procedure to external variables are also resolved when a task is linked. Different tasks may have the same variable defined at different locations.

Consequently, a procedure segment that is shared by more than one task must be coded in such a way as to be independent of the position of its DSEG and of the location of external variables and addresses within a task segment. Code that satisfies this requirement is called DSEG position-independent (DPI) code.

One technique for achieving such independence is to calculate task-dependent addresses using the following:

* A base register that contains the address of a DSEG

* Fixed offsets from that base address to variables and nonlocal addresses

This technique is used in the shared procedure segment S$SYSTEM, a collection of widely used service routines. S$SYSTEM is linked into several tasks in the utility program file, for example, SCI990 and the Text Editor.

This appendix contains a sample conversion using the technique, a set of rules for writing DPI code, and a summary of the conventions followed in S$SYSTEM routines.

B.2 EXAMPLE CONVERSION

As an example, consider the routine S$XMPL shown in Figure B-1. This routine makes the TCA available to the caller. It is written to illustrate several points in a small amount of code. In this context, the code may seem awkward, but the coding techniques are common. The Note indicators at the far right are for reference in the discussion of specific changes to lines of code.

```
*******************************************************
       REF   PARM1               NONLOCAL VARIABLE
       REF   S$NEW               NONLOCAL ADDRESS
       REF   W$XMPL              NONLOCAL ADDRESS
*
       DEF   S$XMPL              ENTRY POINT
*
S$XMPL DATA  W$XMPL,P$XMPL       TRANSFER VECTOR            Note 1
       EVEN
P$XMPL
       MOV   @TCAPTR,R0          TCA INITIALIZED ?          Note 2
       JNE   P$XM80                 -- YES: DO NOTHING
*
       BLWP  @S$NEW              CALL INITIALIZATION        Note 3
       MOV   @PARM1,R0           SEE IF PARAMETERS          Note 4
*                                INDICATE NO "TCA" NEEDED
       JEQ   P$XM80              JUMP IF SO
       BL    @P$SUBR                                        Note 5
       RTWP                      RETURN
P$XM80
       SETO  @TCAPTR             SET FLAG FOR "TCA EXISTS"  Note 2
       CLR   *R13                NORMAL EXIT
       RTWP
*
P$SUBR LI    R4,>0A00            SUBOPCODE=0A & FLAGS=0
       LI    R3,>4300            SVC CODE
       SVC   R3                  DO "GET SEG SIZE" TO SEE IF
       ANDI  R3,>00FF            THE SEGMENT IS THERE
       JEQ   P$SBR5
       LI    R0,>FF05            UNABLE TO ACCESS THE TCA
       MOV   R0,*R13
       RT                        RETURN                     Note 5
P$SBR5 SETO  @TCAPTR             SET FLAG FOR "TCA EXISTS"  Note 2
       CLR   *R13                NORMAL EXIT
       RT                        RETURN                     Note 5
*==========================================================
       DSEG
*----------------------------------------------------------
* DATA SECTION FOR S$XMPL - MUST CONTAIN VOLATILE
*                           INFORMATION
*----------------------------------------------------------
*
*
TCAPTR DATA 0
**********************************************************
```

Notes - Changes to lines are detailed in the corresponding Notes later in this appendix.

Figure B-1 DSEG Position-Dependent Code for S$XMPL

Assume that the object from this code is included in the link stream of task A, as the first routine of the procedure segment X, and that the length of the declared task area for the program is >1000. Figure B-2 illustrates the resolution of addresses in S$XMPL by the linking process.

```
        Procedure                                Task
       Segment                                  Segment
            X                                      A
     +-------------+                        +-------------+
>0000| WP= >2100   |                   >1000| Data area   |
     |-------------|                        | for program |
     | PC= >0004   |                        |      A      |
     |-------------|                        /             /
     |Object of    |                        \             \
     |S$XMPL code, |                   >1FFE|             |
     |with address |         +------>       |-------------|
     |of TCAPTR at |         |         >2000|   TCAPTR    |
     |>2000.       |         |              |-------------|
     |             |         |              /    Other    /
     /             /         |              \     data    \
     \             \         |              |             |
     |             |       DSEGs            |-------------|
     +-------------+        for        >2100| Workspace   |
                        Procedure X         |   W$XMPL    |
                          Routines          |             |
                             |              /             /
                             |              \             \
                             |              |             |
                             |         >211E|             |
                             +------>       +-------------+
```

Figure B-2 Address Resolution in Task Segment A

This resolution of addresses works for task A, and for any replication of task A, since the task segments are identical and TCAPTR, the only volatile data in S$XMPL is in the task segment.

Now assume that program B must perform precisely the same functions that are performed in the routines in procedure segment X. Task B includes the procedure segment in its link stream (as a DUMMY, so that another copy of the object is not generated). The addresses in procedure segment X that are referenced in program B, such as S$XMPL, are resolved. The DSEG elements required by routines in procedure X for volatile data are created at the end of the task segment of program B.

Assume that program B has a declared task area that is >0400 bytes long. Figure B-3 shows the address resolution that results from this linking.

```
                     Task
                    Segment
                       B
              +-------------+
         >1000| Data area   |
              | for program |
              |     B       |
              /             /
              \             \
         >13FE|             |
              |-------------|  <--------+
         >1400|   TCAPTR    |           |
              |-------------|           |
              /    Other    /           |
              \     data    \           |
              |             |         DSEGs
              |-------------|          for
         >1500| Workspace   |      Procedure X
              |   W$XMPL    |        Routines
              |             |           |
              /             /           |
              \             \           |
              |             |           |
         >151E|             |           |
              +-------------+  <--------+
```

Figure B-3 Address Resolution in Task Segment B

When program B executes a BLWP to S$XMPL, the code in procedure segment X expects the variable TCAPTR to be at >2000 and the address of the workspace to >2100! As it is currently written, the location of TCAPTR and the workspace to be used by S$XMPL are fixed in the procedure segment code when the object is linked into task A. This is the image stored on the program file when task A is installed.

In order for the procedure segment code to execute properly, regardless of where the DSEG elements are located in the task segment, references to elements in the task segment must be resolved at run time. This is true whether the elements are variables or addresses. The following modifications accomplish this goal:

* The transfer vector is moved to the DSEG so that the workspace pointer can be resolved properly each time the procedure segment is linked into a different task.

* A register is initialized with the beginning address of the data area for the routine before branching to code in the procedure segment.

* All DSEG position-dependent addresses are referenced by indexing off the base register.

Changes in the lines of code marked with Notes in Figure B-1 are discussed in the paragraphs that follow. Register 10 is used in this example for the base register.

Note 1

Moving the transfer vector and initializing the base register are accomplished by deleting the line at NOTE 1 and adding the following lines in the DSEG:

```
*                        TRANSFER VECTOR
S$XMPL DATA W$XMPL           WORKSPACE POINTER
       DATA D$XMPL           START ADDRESS
*                        INITIALIZE REGISTER 10
D$XMPL LI   R10,BTABLE
*                        BRANCH TO PROCEDURE SEGMENT
       B    @P$XMPL
*                        DEFINE ADDRESS TO BE PUT INTO R10
BTABLE EQU  $
```

Note 2

The address of the variable TCAPTR must be indexed off register 10. The offset of TCAPTR from BTABLE must be known. Define ETCAPT as follows, in the DSEG:

```
ETCAPT EQU  $-BTABLE         OFFSET FROM BTABLE
TCAPTR DATA 0                RESERVE SPACE FOR VARIABLE
```

This provides an offset that can be used in the procedure segment as follows:

```
MOV  @ETCAPT(R10),R0    TCA INITIALIZED?

SETO @ETCAPT(R10)       SET FLAG FOR "TCA EXISTS"

SETO @ETCAPT(R10)       SET FLAG FOR "TCA EXISTS"
```

Note 3

Since S$NEW has a BLWP interface, its transfer vector must also be in the task segment. Therefore, the address of its transfer vector must be indexed off register 10. The address of the transfer vector (S$NEW) is resolved as part of the linking process. The following must appear in the DSEG portion of routine S$XMPL:

```
     E$NEW   EQU  $-BTABLE             OFFSET FROM BTABLE
                  DATA S$NEW           ADDRESS OF S$NEW TRANSFER VECTOR
     *                                 IN THIS TASK SEGMENT
```

The BLWP instruction is replaced with the following instructions:

```
          MOV   @E$NEW(R10),R0      LOAD ADDRESS OF S$NEW TRANSFER VECTOR
          BLWP  *R0                 BRANCH INDIRECT -- TO INITIALIZATION
```

Note 4

PARM1 is a nonlocal variable for S$XMPL, but its address is resolved at link time. The following instructions are added to the DSEG portion, so that the variable can be accessed:

```
     EPARM1 EQU  $-BTABLE              OFFSET FROM BTABLE
                 DATA PARM1            ADDRESS OF PARM1
```

The MOV instruction is replaced with the following instructions:

```
          MOV   @EPARM1(R10),R0     LOAD ADDRESS OF PARM1
          MOV   *R0,R0              LOAD INDIRECT
```

Note 5

If subroutine P$SUBR is to be called by more than one module in procedure segment X, the interface to the subroutine P$SUBR requires modification.

Assume that a routine in another module, say S$EX2, calls the subroutine P$SUBR. When P$SUBR is called by S$EX2, register 10 is the base address of the S$EX2 data area, but P$SUBR must execute using its local DSEG in order to modify the value of TCAPTR. This can be accomplished by meeting the following interface requirements in the subroutine:

* The entry point must be in the DSEG.

* R10 of the calling routine is saved and the local BTABLE address of the subroutine is placed in R10 prior to executing code in the procedure segment.

* R10 of the caller is restored prior to the return.

The requirements on the calling routine are:

* The entry must be declared in the DSEG.

* The calling routine must branch indirect to the subroutine.

The following changes make it possible for any other routine in procedure segment X to call P$SUBR:

Call the entry point S$SUBR, for sake of consistency. Make the following additions to the BTABLE structure:

```
E$SAVE  EQU  $-BTABLE       OFFSET FROM BTABLE
        DATA 0              RESERVE SPACE FOR SAVING R10
E$SUBR  EQU  $-BTABLE       OFFSET FROM BTABLE
        DATA S$SUBR         ADDRESS OF ENTRY POINT IN TASK
```

Define the entry point and preprocessing in the DSEG as follows:

```
*                           SAVE CURRENT R10
S$SUBR  MOV R10,@BTABLE+E$SAVE
*                           LOAD LOCAL BTABLE ADDRESS
        LI  R10,BTABLE
*                           BRANCH TO PROCEDURE CODE
        B   @P$SUBR
```

R10 of the caller must be restored before the return. The following instruction is inserted before each of the RT statements in P$SUBR.

```
        MOV @E$SAVE(R10),R10
```

The call to P$SUBR in S$XMPL must be compatible with the newly defined interface. The BL instruction must be replaced as follows:

```
        MOV @E$SUBR(R10),R0     ADDRESS OF ENTRY POINT
        BL  *R0                 BRANCH INDIRECT
```

S$SUBR can now be called by any routine in procedure segment X, using the calling sequence just shown.

With these changes, S$XMPL can be linked with either task and execute properly for each.

The object from code shown in Figure B-4 is the portion of the final version of S$XMPL that appears in the shared procedure segment. Figure B-5 shows the code from which the S$XMPL DSEG in the task segment is generated. The REF and DEF instructions are moved to the DSEG to improve readability of the listing. This keeps them near the BTABLE where they are used.

```
       EVEN
P$XMPL
       MOV  @ETCAPT(R10),R0    TCA INITIALIZED ?
       JNE  P$XM80                -- YES: DO NOTHING
*
       MOV  @E$NEW(R10),R0     LOAD ADDRESS OF S$NEW TRANSFER VECTOR
       BLWP *R0                BRANCH INDIRECT-TO INITIALIZATION
       MOV  @EPARM1(R10),R0    LOAD ADDRESS OF PARM1
       MOV  *R0,R0             LOAD INDIRECT-SEE IF PARAMETERS
*                              INDICATE NO "TCA" NEEDED
       JEQ  P$XM80             JUMP IF SO
       MOV  @E$SUBR(R10),R0    ENTRY POINT OF S$$SUBR IS IN DSEG
       BL   *R0                BRANCH INDIRECT TO S$SUBR
       RTWP                    RETURN
P$XM80
       SETO @ETCAPT(R10)       SET FLAG FOR "TCA EXISTS"
       CLR  *R13               NORMAL EXIT
       RTWP
*
P$SUBR LI   R4,>0A00           SUBOPCODE=0A & FLAGS=0
       LI   R3,>4300           SVC CODE
       SVC  R3                 DO "GET SEG SIZE" TO SEE IF
       ANDI R3,>00FF           THE SEGMENT IS THERE !
       JEQ  P$SBR5
       LI   R0,>FF05           UNABLE TO ACCESS THE TCA
       MOV  R0,*R13
       MOV  @E$SUBR(R10),R10   RESTORE CALLER'S R10
       RT                      RETURN
P$SBR5 SETO @ETCAPT(R10)       SET FLAG FOR "TCA EXISTS"
       CLR  *R13               NORMAL EXIT
       MOV  @E$SUBR(R10),R10   RESTORE CALLER'S R10
       RT                      RETURN
*****************************************************************
```

Figure B-4 S$XMPL Code for Procedure Segment

```
*=================================================================
       DSEG
*-----------------------------------------------------------------
* DATA SECTION FOR S$XMPL - MUST CONTAIN ALL RELOCATABLE
*                             INFORMATION
*-----------------------------------------------------------------
*
       REF  PARM1          NONLOCAL VARIABLE
       REF  S$NEW          NONLOCAL ADDRESS
       REF  W$XMPL         NONLOCAL ADDRESS
*
       DEF  S$XMPL         ENTRY POINT
*
*                          TRANSFER VECTOR
S$XMPL DATA W$XMPL         WORKSPACE POINTER
       DATA D$XMPL         START ADDRESS
*                          INITIALIZE REGISTER 10
D$XMPL LI   R10,BTABLE
                           BRANCH TO PROCEDURE SEGMENT
       B    @P$XMPL
*                          ENTRY FOR P$SUBR
S$SUBR MOV  R10,@BTABLE+E$SAVE     SAVE CALLER'S REGISTER 10
       LI   R10,BTABLE             LOAD LOCAL BTABLE ADDRESS
       B    @P$SUBR                BRANCH TO PROCEDURE SEGMENT
*                          DEFINE ADDRESS TO BE PUT INTO R10
BTABLE EQU  $
*
ETCAPT EQU  $-BTABLE       OFFSET FROM BTABLE
TCAPTR DATA 0              RESERVE SPACE FOR VARIABLE
*
EPARM1 EQU  $-BTABLE       OFFSET FROM BTABLE
       DATA PARM1          ADDRESS OF PARM1
*
E$NEW  EQU  $-BTABLE       OFFSET FROM BTABLE
       DATA S$NEW          ADDRESS OF S$NEW TRANSFER VECTOR
*                          IN THIS TASK SEGMENT
*
E$SUBR EQU  $-BTABLE       OFFSET FROM BTABLE
       DATA S$SUBR         ADDRESS OF S$SUBR IN TASK SEGMENT
*
E$SAVE EQU  $-BTABLE       OFFSET FROM BTABLE
       DATA 0              RESERVE SPACE FOR SAVING CALLER'S R10
******************************************************************
```

Figure B-5 S$XMPL Code for Task Segment

Figure B-6 shows the resolution of the addresses when procedure X, containing this routine, is linked into task A and into task B.

```
        Program B                              Program A
          Task                                   Task
         Segment                                Segment
     +-------------+                        +-------------+
>1000| Data area   |                   >1000| Data area   |
     | for program |                        | for program |
     |     B       |                        |     A       |
     /             /                        /             /
     \             \                        \             \
>13FE|             |                        |             |
     |-------------|<----+                  |             |
>1400| WP= >1600   |     |                  |             |
     |-------------|     |                  |             |
     | PC= >1404   |     |             >1FFE|             |
     |-------------|     |         +------> |-------------|
>1404| LI R10,>140C|     |         |   >2000| WP= >2200   |
     |-------------|     |         |        |-------------|
     | B @P$XMPL   |     |         |        | PC= >2004   |
     |-------------|     |         |        |-------------|
>140C|   TCAPTR    |     |         |   >2004| LI R10,>200C|
     |-------------|     |         |        |-------------|
     |PARM1 address|     |         |        | B @P$XMPL   |
     |-------------|     |         |        |-------------|
     |S$NEW address|     |         |   >200C|   TCAPTR    |
     |-------------|   DSEGs for Procedure  |-------------|
     /             /     X  Routines        |PARM1 address|
     \             \     |         |        |-------------|
     |-------------|     |         |        |S$NEW address|
>1600|  Workspace  |     |         |        |-------------|
     |    W$XMPL   |     |         |        /             /
     |             |     |         |        \             \
     /             /     |         |        |-------------|
     \             \     |         |   >2200| Workspace   |
     |             |     |         |        |   W$XMPL    |
>161E|             |     |         |        |             |
     +-------------+<----+         |        /             /
                                   |        \             \
                                   |   >221E|             |
                                   +-------->+-------------+
```

Figure B-6 DPI Task Structures

Notice that in both cases the values of the offsets, ETCAPT, EPARM1, and E$NEW, are the same, but the address that is passed to S$XMPL in register 10 is different. The offsets are hard coded in the procedure segment, but every task that includes the procedure segment X produces the identical DSEG structure, so that the offsets are valid, regardless of the position of the DSEG within the task segment.

B.3 RULES

The rules for implementing this technique for writing DSEG position-independent code are summarized as follows:

* The transfer vector for a routine with BLWP/RTWP interface must be in the DSEG.

* The entry point of a common subroutine with BL/RT interface must be defined in a DSEG.

* All local variables must be accessed by indexing off the base register.

* All nonlocal variables and addresses must be accessed through a local variable that contains the address of the item.

* If a BL/RT interface is used for a common routine, the called routine must save/restore the base register.

B.4 CONVENTIONS USED IN S$SYSTEM

S$SYSTEM is a procedure segment that is linked into several tasks in the utility program file. All routines in S$SYSTEM are DSEG position-independent.

All S$SYSTEM routines have a BLWP/RTWP interface. Five workspaces are defined for use by S$ routines. The routines are mapped into the workspaces by aliases with the format W$name, where name resembles the name of the routine.

All S$SYSTEM routines use register 10 in their own workspace for the address of the BTABLE structure. (Counting down from register 15, R10 is the first uncommitted register. R13 through R15 are used by BLWP/RTWP, R12 by read/write CRU instructions, and R11 by BL/RT.)

Figure B-7 shows the structure of a typical DSEG for a S$SYSTEM routine.

```
*****************************************************************
*                        LOCAL VARIABLES, EXTERNAL ADDRESSES
*        DEF
*        REF
*
*                        TRANSFER VECTOR
S$name   DATA    W$name         S$ routine workspace pointer
         DATA    D$name         S$ routine program counter
*
D$name   LI      R10,BTABLE     Put BTABLE address in register 10
         B       @P$name        Entry point in procedure segment
*                        DATA AREA POINTED TO  BY R10
BTABLE   EQU     $
*      Reserve space for all local variables
*      Declare all nonlocal addresses
*****************************************************************
```

Figure B-7 DSEG Structure in S$SYSTEM Routines

The procedure segment S$SYSTEM is installed with write protection.

The convention of using a label of E followed by the (possibly abbreviated) name of the variable is generally followed in SSYSTEM routines. This makes it easier to recognize what is being referenced when reading the code and/or a cross reference listing.

All routines in S$SYSTEM have names that begin with S$, but such names are not reserved in DNOS for DSEG position independent code. The name of a routine cannot be assumed to imply DSEG position independence.

APPENDIX C

TASK, PROCEDURE AND OVERLAY SEGMENTS IN S$UTIL

C.1 OVERVIEW

This appendix lists the task IDs, overlay IDs and procedure segment IDs associated with SCI and DNOS utility programs. All elements listed in this appendix are installed in the program file S$UTIL, except AGTASK which is in S$SECURE.

Table C-1 provides a list of the task IDs, including the name of each task and its function. The tasks documented in this manual are grouped by subsystem. The tasks listed as miscellaneous tasks are not documented in this manual. Operating system tasks are not shown in this list. They are described in the DNOS System Design Document.

Table C-2 provides a list of the procedure/program segments, the installed IDs and names, and the purpose of the segment.

Table C-3 provides a list of the overlays, the IDs and names and the purpose of the overlay. They are grouped by the task to which they belong.

Table C-1 SCI/Utilities Tasks

Subsystem	ID	Name	Purpose
SCI990	01	SCI990	The System Command Interpreter (SCI)
Text Editor	0E	EDITOR	The Text Editor
User ID Maintenance			
	39	AUIDUI	Add/delete a user ID
	09	MPC	Modify Passcode
	02	AGTASK	Access Group Maintenance (in S$SECURE)
File Maintenance			
	11	CCAF	Copy or append one file to another file
	13	RD	Restore a directory
	14	VB	Verify a backup of a directory
	22	DD	Delete a directory
	23	LD	List the contents of a directory structure
	32	MD	Map a disk or a directory
	34	CD	Copy a directory to another director
	35	BD	Back up a directory on a sequential medium
	36	VC	Verify a copy of a directory
	40	MKL	Modify key logging
Operator Interface			
	61	OPERATOR	The Operator Interface system task
	62	XOI	The Operator Interface user interfac
	63	LGRCRT	Create system log files
System Configuration Utility			
	2E	SCU	Performs SCU functions
Spooler			
	4D	SP$DST	Schedule Spooler devices, maintain Spooler queues
	4E	SPINIT	SP$DST initialization
	57	PF	Spooler user interface
	59	LPWRITER	Line printer writer
	5B	SPTASK	Interface between a user task and SP$DST
	60	SOS	Show status of output queues
Message File Maintenance			
	3C	BMF	Build a message file
	49	BEMF	Build an expanded message file
	55	SEM	Show an expanded message

Table C-1 SCI/Utilities Tasks (Continued)

Subsystem	ID	Name	Purpose
Miscellaneous Task Segments			
	02	TINFO	Access terminal information
	03	MS	Modify synonym
	07	MAILBOX	Perform message routing functions
	08	CKD	Check disk consistency
	0D	DEBUGGER	Perform Debugger functions
	0E	EDITOR	Text Editor
	0F	TIGR	Interactive execution of SVCs
	10	MRFSRF	Modify/show relative record file
	12	LS	List synonyms
	15	CP	Create patch
	18	RVI	Recover volume information (Track 0, Sector 0 unusable)
	1A	ANALZ	Analyze a crash dump
	1B	IFSVC	Collection of SVC execution routines
	1C	XBJ	Execute batch job
	1D	MPFMKF	Map program file or key indexed file (KIF)
	24	LLR	List logical record
	26	MPISPI	Modify/show program image file
	2A	IDT	Initialize date and time
	2C	MADSAD	Modify/show absolute disk
	2F	SND	Write logical names and synonyms to disk
	33	SYSGEN	Generate an operating system
	3E	MOEMPE	Modify overlay/procedure entry
	3F	CPI	Copy program image file
	45	CSKCKS	Copy sequential file to KIF or KIF to sequential file
	46	RWCRU	Read/write at a CRU address
	4B	IBMUTL	Convert flexible disk from IBM to TI format, and vice versa
	4F	ASP	Assign spooler parameters
	5C	ALN	Assign logical name
	5D	DCOPY	Track-by-track copy (disk-to-disk)
	5E	CSM	Copy sequential medium to sequential medium
	5F	SDA	Show device attributes
	64	LSC	List software configuration
	65	CB	Create batch stream
	66	SRFI	Show relative record file (interactively)

Table C-2 SCI/Utilities Procedure Segments

ID	Name	Purpose
02	S$SYSTEM	S$ utility routines
03	SCI990	System Command Interpreter (SCI)
04	TIGR	Interactive execution of SVCs
05	EDITOR	Text Editor
06	SPCOMN	Common data for Spooler subsystem
07	LPWRITER	Line printer writer

Table C-3 SCI/Utilities Overlays

Subsystem	ID	Name	Purpose
Debugger			
	07	L$$PB3	Debugger
	08	L$$PB4	Debugger
	09	L$$PB5	Debugger
	0A	L$$RS3	Debugger
	0B	L$$RS4	Debugger
	0D	L$$AB	Debugger
	0E	L$$ASB	Debugger
	0F	L$$DEB	Debugger
	10	L$$FB	Debugger
	11	L$$MI	Debugger
	12	L$$MM	Debugger
	13	L$$PB	Debugger
	14	L$$RS	Debugger
	15	L$$PB1	Debugger
	16	L$$PB2	Debugger
	17	L$$RS1	Debugger
	18	L$$RS2	Debugger
	19	L$$DB	Debugger
	1A	L$$MR	Debugger
	1B	L$$RS5	Debugger
	50	L$$SPS	Debugger
	51	L$$PB6	Debugger
	52	L$$APB	Debugger
System Configuration Utility			
	41	SCUINIT	Initialization, termination
	42	SCUDEV	Modify device configuration
	43	SCUADD	Add device
	44	SCUPDT	Build PDT
	45	SCUDSR	Install DSR, add interrupt
	46	SCUDEL	Delete device, modify device state
	47	SCUMISC	Modify system parameters, tables Initialize system log
System Generation			
	01	INIT	Initialization
	02	INTERACT	User interface
	03	BUILD	Generate system

ALPHABETICAL INDEX

Introduction

The following index lists key words and concepts from the subject material of this manual together with the area(s) in the manual that supply coverage of the listed concept. The numbers along with the right side of the listing reference the following manual areas:

* Sections -- References to Sections of the manual appear as "Section x" with the symbol x reresenting any numeric quantity.

* Appendixes -- References to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

* Paragraphs -- References 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 chartacter refers to the section or appendix of the manual in which the paragraph is 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

Should you be unable to find the item of interest in the index, review the Table of Contents, List of Tables and List of Figures for general categories of information.

Alphabetical Index

USER'S RESPONSE SHEET

CUT ALONG LINE

Manual Title: DNOS SCI and Utilities Design Document (2270513-9701)

Manual Date: 15 November 1983 Date of This Letter: ______

User's Name: ______ Telephone: ______

Company: ______ Office/Department: ______

Street Address: ______

City/State/Zip Code: ______

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual	Comment/Suggestion

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
DIGITAL SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD