Application Programming Guide

Part No. 946250-9703 *F
January 1985

Volume III

TEXAS INSTRUMENTS

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES AND DISCARD SUPERSEDED PAGES

Note: The changes in the text are indicated by a change number at the bottom of the page and a vertical bar in the outer margin of the changed page. A change number at the bottom of the page but no change bar indicates either a deletion or a page layout change.

DX10 Operating System Application Programming Guide, Volume III (946250-9703)

Original Issue .... August 1977
Revision .... March 1978
Revision .... October 1978
Revision .... December 1979
Revision .... April 1981
Revision .... September 1982
Revision .... September 1983
  Change 1 .... January 1985

Total number of pages in this publication is 436 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	1	2-1 - 2-7	0	3-16 - 3-18	1
Effective Pages	1	2-8	1	3-19 - 3-28	0
Eff. Pages Cont.	1	2-9 - 2-14	0	4-1 - 4-6	0
iii - iv	1	2-15 - 2-16	1	4-7	1
v - xv	0	2-17 - 2-20	0	4-8 - 4-12	0
xvi	1	3-1 - 3-13	0	5-1 - 5-14	0
xvii - xx	0	3-14	1	6-1 - 6-32	0
1-1 - 1-2	0	3-15	0	6-33 - 6-34	1

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

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

© 1977, 1978, 1979, 1981, 1982, 1983, 1985, Texas Instruments Incorporated. All Rights Reserved.

Printed in U.S.A.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Texas Instruments Incorporated.

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES AND DISCARD SUPERSEDED PAGES

Note: The changes in the text are indicated by a change number at the bottom of the page and a vertical bar in the outer margin of the changed page. A change number at the bottom of the page but no change bar indicates either a deletion or a page layout change.

DX10 Operating System Application Programming Guide, Volume III (946250-9703)

Continued:

PAGE NO.	CHANGE NO.
6-35 - 6-50	0
6-51	1
6-52 - 6-66	0
6-67/6-68	1
7-1 - 7-8	0
8-1 - 8-3	0
8-4	1
8-5 - 8-29	0
8-30	1
8-31 - 8-36	0
9-1 - 9-30	0
9-31	1
9-32 - 9-40	0
10-1 - 10-27	0
10-28 - 10-29	1
10-30 - 10-49	0
10-50	1
11-1 - 11-10	0
11-11 - 11-16	1
11-16A - 11-16B	1
11-17 - 11-18	1
11-18A - 11-18B	1
11-19	0
11-20 - 11-21	1
11-22	0
11-22A - 11-22B	1
11-23 - 11-36	0
12-1 - 12-4	0
12-5	1
12-6 - 12-10	0
A-1 - A-14	0
B-1 - B-11	0
B-12	1
B-13 - B-35	0
B-36	1
C-1/C-2	0
D-1 - D-10	0
E-1 - E-6	0
F-1 - F-2	0
Index-1 - Index-10	0
User's Response	1
Business Reply	1
Inside Cover	1
Cover	1

DX10 Software Manuals

DX10 Operating System Manuals

DX10 Operating System
Concepts and Facilities
(Volume I)
946250-9701

DX10 Operating System
Operations Guide
(Volume II)
946250-9702

DX10 Operating System
Application Programming
Guide (Volume III)
946250-9703

DX10 Operating System
Text Editor Manual
(Volume IV)
946250-9704

DX10 Operating System
Systems Programming
Guide (Volume V)
946250-9705

DX10 Operating System
Error Reporting and
Recovery Manual
(Volume VI)
946250-9706

DX10 Operating System
Release 3.7 System
Design Document
939153-9701

Link Editor Reference
Manual
949617-9701

Communications Manuals

DX10 3270 Interactive
Communications
Software (ICS) User's
Guide
2250954-9701

DX10 Remote Terminal
Subsystem (RTS) System
Generation and
Programmer's Reference
Manual
2272054-9701

DX10 Remote Terminal
Subsystem (RTS)
Operator's Guide
2272055-9701

DX10 3780/2780 Emulator
Release 4 User's Guide
2302683-9701

Language Manuals

990/99000 Assembly
Language Reference
Manual
2270509-9701

DX10 COBOL
Programmer's Guide
2270521-9701

COBOL Reference Manual
2270518-9701

DX10 FORTRAN-78
Programmer's Guide
2268679-9701

FORTRAN-78 ISA
Extensions Manual
2268696-9701

FORTRAN-78
Reference Manual
2268681-9701

Report Program
Generator (RPG II)
Programmer's Guide
939524-9701

TI BASIC Reference
Manual
2308769-9701

DX10 TI Pascal
Programmer's Guide
2270528-9701

TI Pascal Reference
Manual
2270519-9701

TI Pascal Configuration
Processor Tutorial
2250098-9701

Miscellaneous Software Manuals

Operator's Guide,
Business System 300
(International)
2533318-9701

Operator's Guide,
Business System 300
(Domestic)
2533318-9702

ROM Loader User's
Guide
2270534-9701

Operator's Guide,
Business System 300A
2240275-9701

Productivity Tools Manuals

DX10 Data Base
Management System
Programmer's Guide
2250425-9701

DX10 Data Base
Administrator User's
Guide
2250426-9701

TIFORM
Reference Manual
2234391-9701

DX10 Query-990 User's
Guide
2250466-9701

Model 990 Computer DX
Sort/Merge User's Guide
946252-9701

DX10 TIPE Texas
Instruments Page
Editor Reference
Manual
2302656-9701

Data Dictionary User's
Guide
2276582-9701

DX10 COBOL Program
Generator User's
Guide
2308956-9701

DX10 Hardware Manuals

Miscellaneous Hardware Manuals

- Model 990 Computer Communications System Installation and Operation 945409-9701
- Model 990 Computer PROM Programming Module Installation and Operation 945258-9701
- Model 990 Computer TILINE Coupler User's Guide 2268688-9701
- 990 CRU/TILINE Expansion Installation and Operation 2272075-9701
- Model 990 Computer Remote Terminal Subsystem (RTS) Hardware Installation Manual (European Edition) 2272053-9701
- Model 990 Computer Remote Terminal Subsystem (RTS) Hardware Installation Manual (Domestic Edition) 2272053-9702
- Model 990/10 Computer System Hardware Reference Manual 945417-9701
- ROM Loader User's Guide 2270534-9701

Hard-Copy Terminal Manuals

- Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation 945259-9701
- Model 990 Computer Model 743 KSR Data Terminal Installation and Operation 943462-9701
- Model 743 KSR Terminal Operator's Manual 984030-9701
- Model 745 Portable Terminal Operator's Manual 984024-9701
- Models 763/765 Operating Instructions 2203664-9701
- Models 763/765 Memory Terminals Systems Manual 2203665-9701
- Model 783 KSR Terminal Operator's Manual 2265936-9701
- Model 785 Communications Terminal Operator's Manual 2265937-9701
- Model 787 Communications Terminal Operator's Manual 2265938-9701
- Model 820 KSR Terminal Operator's Manual 2208225-9701
- Model 990 Computer Model 820 KSR Data Terminal Installation and Operation 2250454-9701
- Model 990 Computer Model 840 RO Printer Installation and Operation Manual 2302695-9701
- Operator's Guide, Model 840 RO Printer Business System Series 2533270-9701

Display Terminal Manuals

- Model 931 Video Display Terminal Installation and Operation 2229228-0001
- Model 990 Computer Model 940 Electronic Video Terminal (EVT) Installation and Operation Manual 2250368-9701
- Model 990 Computer Model 911 Video Display Terminal Installation and Operation 945423-9701
- Model 990 Computer Model 913 CRT Display Terminal Installation and Operation 943457-9701

Printer Manuals

- Model 990 Computer Models 306 and 588 Line Printers Installation and Operation 945261-9701
- Model 781 RO Terminal Operator's Manual 2265935-9701
- Model 990 Computer Model 810 Printer Installation and Operation 939460-9701
- Operator's Guide, Model 810 Printer Business System Series 2228256-9701
- Model 820 KSR Terminal Operator's Manual 2208225-9701
- Model 990 Computer Model 820 KSR Data Terminal Installation and Operation 2250454-9701
- Model 990 Computer Model 840 RO Printer Installation and Operation Manual 2302695-9701
- Operator's Guide, Model 840 RO Printer Business System Series 2533270-9701
- Model 850 Printer User's Manual 2219890-0001
- Model 990 Computer Model 2230 and 2260 Line Printers Installation and Operation 946256-9701
- Model 990 Computer Model LP300 and LP600 Line Printers Installation and Operation Manual 2250364-9701
- Models LP300 and LP600 Line Printers Installation and Operation (Business System Series) 2302643-9701
- Model 990 Computer Model LQ45 Letter Quality Printer System Installation and Operation Manual 2268695-9701
- Model LQ55 Letter Quality Printer Installation and Operation 2234382-9701

Storage Device Manuals

- Model CD1400 Disk System Installation and Operation Manual 2272081-9701
- Rectilinear CD1400 Disk System Installation and Operation (Business System Series) 2311346-9701
- Model 990 Computer Model DS10 Cartridge Disc System Installation and Operation 946261-9701
- Model 990 Computer Model DS25/DS50 Disc Systems Installation and Operation 946231-9701
- Model 990 Computer Moving Head Disc System Installation and Operation 945260-9701
- WD500/WD500A Mass Storage System Installation and Operation (Business System Series) 2302688-9701
- Model 990 Computer Model DS80 Disk System Installation and Operation Manual 2302629-9701
- Model 990 Computer Model DS200 Disc System Installation and Operation 949615-9701
- Model DS300 Disk System Installation and Operation Manual 2302631-9701
- Operator's Guide Model WD500/WD500A Disk Unit (Business System Series) 2533269-9701
- Operator's Guide Model WD800/WD800A Disk Unit (Business System Series) 2533319-9701
- WD800/WD800A Mass Storage System Installation and Operation 2306140-9701
- Model 990 Computer Model FD800 Floppy Disc System Installation and Operation 945253-9701
- Model 990 Computer Model FD800 Floppy Disc System with International Chassis Installation and Operation Manual 2250697-9701
- 990 TILINE Floppy Disc Installation and Operation 2261886-9701
- Model 990 Computer Model FD1000 Flexible Disk System with International Chassis Installation and Operation 2250698-9701
- Model 990 Computer Model 979A Magnetic Tape System Installation and Operation 946229-9701
- Model MT1600 Magnetic Tape System Installation and Operation 2302642-9701
- Model 990 Computer Model 804 Card Reader Installation and Operation 945262-9701
- WD900/MT3200 General Description Manual 2234398-9701

Preface

This manual provides general information about how DX10 handles programs, the DX10 input/output (I/O) system, programming considerations for the DX10 operating system, complete descriptions of available supervisor calls (SVCs), and information on using the Debugger for assembly language programs. This manual provides the framework that allows you to understand how DX10 functions support applications programs. With this understanding, you can design and implement applications programs in any language supported by DX10.

This is one of a set of six volumes that describe the operational characteristics and features of DX10. In addition to the six volumes, several support manuals are available for DX10 functions. Also, each language supported by DX10 has its own associated manuals.

Become acquainted with these volumes and related DX10 manuals as necessary to prepare and execute application programs on DX10. The following paragraphs contain a brief comment regarding the contents of the other volumes in the set. (The full titles and part numbers of all manuals associated with the DX10 operating system are provided in the frontispiece.)

Concepts and Facilities (Volume I) includes features, concepts, and general background information describing the DX10 operating system. It also contains a master subject index to help you find the information you need.

The *Operations Guide* (Volume II) contains information on how to perform an initial program load (IPL start procedure) and how to log on and operate a terminal. Additionally, this manual contains an introduction to your interface with DX10, the System Command Interpreter (SCI), and includes a complete description of the SCI commands required to operate DX10. (The Text Editor and Link Editor commands are not included in Volume II, but can be found in their respective manuals. Debugger commands are in Volume III.)

The *Text Editor Manual* (Volume IV) includes operating instructions, examples, and exercises for the interactive Text Editor provided on DX10. The SCI commands and error messages related to the Text Editor are included.

The *Systems Programming Guide* (Volume V) includes information required by the system programmer to maintain and extend a computer system running under DX10. The disk build procedure required for building your initial system disk, and the system generation procedure and troubleshooting guide required for system start-up are located in this manual. The manual also includes support of nonstandard devices and the privileged SVCs available on DX10.

The *Error Reporting and Recovery Manual* (Volume VI) describes each error message you can receive while operating DX10, and gives suggested procedures for recovery. It documents task errors, command errors, SVC errors, SCI errors, and I/O errors including those from disk and magnetic tape. Also included are sections on system crash analysis, and system troubleshooting.

NOTE

Additional, in-depth descriptions related to specific languages including FORTRAN, COBOL, BASIC, RPG II, TI Pascal, assembly language, and Query are found in manuals dedicated to the appropriate programming language. A Link Editor manual is provided as a separate volume describing the link edit function in a DX10 environment. Separate manuals describe the use of an optional Sort/Merge package and the DBMS package.

Contents

Paragraph	Title	Page

1 — Introduction

1.1	General	1-1
1.2	Applications Programming on DX10	1-1

2 — How DX10 Manages Programs

2.1	Introduction	2-1
2.2	Program Structure	2-1
2.2.1	Program Segmentation	2-2
2.2.2	Program Mapping	2-3
2.2.2.1	Implementing Program Segmentation	2-5
2.2.2.2	Task Segment Only	2-5
2.2.2.3	Task Segment and System Common Segment	2-5
2.2.2.4	Task Segment and One or Two Procedure Segments	2-6
2.2.2.5	Task Segment, Procedure Segment, and System Common	2-7
2.2.2.6	Overlays	2-7
2.3	Task Execution	2-7
2.4	How DX10 Manages Task Execution	2-8
2.4.1	Task Scheduling	2-8
2.4.1.1	Priority	2-9
2.4.1.2	Task Sentry	2-10
2.4.1.3	How Priority Scheduling Affects Applications	2-10
2.4.2	Dynamic Memory Management	2-10
2.5	Programming Considerations	2-11
2.5.1	Reentrancy and Sharing Code	2-11
2.5.1.1	Address-Independent Reentrant Procedures	2-12
2.5.1.2	Address-Dependent Reentrant Procedures	2-12
2.5.1.3	Data, or "Dirty" Procedures	2-12
2.5.2	Sharing Data in Memory	2-13
2.5.2.1	Restrictions on Using Shared Data	2-13
2.5.2.2	Cautions to Observe	2-14
2.5.3	Using the Intertask Communication (ITC) Channels	2-15
2.5.4	Sharing Procedure Code to Save Memory	2-15
2.5.5	User Program Files	2-15
2.5.6	Overlays	2-16
2.5.6.1	Overlay Structures	2-16
2.5.6.2	Overlay Loading	2-16
2.5.6.3	Relocatable Overlays	2-17
2.5.7	Task Attributes	2-17

Paragraph	Title	Page
2.5.7.1	Privileged and Nonprivileged Tasks	2-17
2.5.7.2	System and User Tasks	2-17
2.5.7.3	Priority	2-18
2.5.7.4	Disk Resident and Memory Resident Tasks	2-18
2.5.7.5	Replicatable and Nonreplicatable Tasks	2-18
2.5.7.6	Arithmetic Overflow Protection	2-18
2.5.7.7	Execute Protection	2-18
2.5.8	Programming Prohibitions	2-18
2.6	Task Termination	2-19
2.6.1	Normal Termination	2-19
2.6.2	Abnormal Termination	2-20

3 — DX10 I/O System

3.1	Introduction	3-1
3.2	Supported File Types and Usage	3-1
3.2.1	Sequential Files	3-1
3.2.2	Relative Record Files	3-2
3.2.3	Key Indexed Files	3-3
3.2.4	File Usage Tradeoffs	3-4
3.2.4.1	Sequential Files Versus Relative Record Files	3-4
3.2.4.2	Relative Record Files Versus KIFs	3-5
3.3	File Features	3-6
3.3.1	Blank Suppression and Adjustment	3-6
3.3.2	Expandable Files	3-7
3.3.3	End-of-File (EOF)	3-7
3.3.4	File and Record Protection Features	3-7
3.3.4.1	Delete and Write Protection	3-8
3.3.4.2	Record Locking	3-8
3.3.4.3	File Access Privileges	3-9
3.3.4.4	Immediate or Forced Write	3-10
3.3.4.5	Special Usage File Protection	3-11
3.3.5	How the System Handles File I/O	3-11
3.4	Disk File Organization and Management	3-11
3.4.1	File Management Strategy	3-12
3.4.2	Physical Disk Structure	3-13
3.4.2.1	Disk Sectors	3-13
3.4.2.2	Allocatable Disk Units (ADUs)	3-13
3.4.2.3	Format Information for Supported Disks	3-13
3.4.3	File Structure	3-14
3.4.4	Logical Records	3-14
3.4.4.1	Constraints for Sequential Files	3-15
3.4.4.2	Constraints for Relative Record Files	3-16
3.4.4.3	Constraints on Key Indexed Files	3-17
3.4.4.4	Choosing Logical Record Length	3-18
3.4.5	Physical Records	3-18
3.4.5.1	Choosing Physical Record Size	3-19
3.4.5.2	Default Physical Record Size	3-19

Paragraph	Title	Page
3.4.6	Blocking and Blocking Buffers	3-20
3.4.6.1	Choosing Logical Records per Physical Record	3-20
3.4.7	Unblocked Files	3-22
3.4.8	How DX10 Allocates Disk File Space	3-22
3.4.8.1	Secondary Allocation Algorithm	3-23
3.4.8.2	Exception to the Algorithm	3-23
3.5	Logical Unit Numbers (LUNOs) and Devices	3-24
3.5.1	I/O Device Access Through LUNOs	3-25
3.5.1.1	Global or Station Local LUNO Strategy	3-25
3.5.1.2	Task Local LUNO Strategy	3-25
3.5.2	Access Logic for Devices	3-27
3.5.2.1	Record-Oriented Devices	3-27
3.5.2.2	File-Oriented Devices	3-27
3.5.3	Using the Printer and Printer Files	3-27

4 — Designing Applications for Data Protection

Paragraph	Title	Page
4.1	Introduction	4-1
4.2	Error Reporting	4-1
4.3	Transaction Logging Principles	4-2
4.3.1	When to Use Transaction Logging	4-2
4.3.1.1	Cost of Implementation	4-3
4.3.1.2	Computing Mean Time Between Failures	4-3
4.3.1.3	The Log File as Audit Trail	4-4
4.3.2	Designing the Recovery Procedures	4-4
4.3.2.1	Step 1: Backup	4-5
4.3.2.2	Step 2: Activity Interval	4-6
4.3.2.3	Step 3: Archive the Transaction Log	4-6
4.3.2.4	Step 4: Recovery	4-6
4.3.3	Blocking Log Entries	4-7
4.3.4	File Type for Transaction Log	4-9
4.3.5	Tape File Considerations	4-10

5 — Programming with Assembly Language on DX10

Paragraph	Title	Page
5.1	Introduction	5-1
5.2	General Programming Considerations	5-2
5.2.1	Assembly Language Program Segmentation	5-2
5.2.2	Attaching Procedures to Tasks	5-3
5.2.3	Transfer Vector	5-3
5.2.4	End Action Routines	5-3
5.2.5	Using SVCs	5-4
5.2.6	Using Subroutines	5-4
5.2.7	Using Overlays	5-5
5.3	Programming with Assembly Language	5-5
5.3.1	Writing Assembly Language Programs	5-5
5.3.2	Assembling Assembly Language Programs	5-6

Paragraph	Title	Page
5.3.3	Linking Assembly Language Programs	5-7
5.3.4	Installing Assembly Language Programs	5-7
5.3.5	Executing Assembly Language Programs	5-8
5.3.6	Assembly Language Programming Exercise	5-8

6 — SCI Programming Language

Paragraph	Title	Page
6.1	Introduction	6-1
6.1.1	What an SCI Command Procedure Is	6-1
6.1.2	What a Procedure Directory Is	6-2
6.1.3	What an SCI Command Processor Is	6-2
6.1.4	What a Processor Subroutine Is	6-3
6.1.5	What the SCI Language Is	6-3
6.2	Implementing Command Procedures, Processors, and Menus	6-4
6.2.1	Creating and Changing SCI Command Procedures	6-4
6.2.1.1	Using the Text Editor	6-4
6.2.1.2	Creating Command Procedures Interactively	6-5
6.2.1.3	Using a Batch Stream	6-5
6.2.1.4	Creating a Procedure Directory	6-5
6.2.1.5	Naming New Command Procedures	6-6
6.2.2	Creating and Changing SCI Command Processors	6-6
6.2.3	Creating and Changing SCI Command Menus	6-7
6.2.4	Log-On/Log-Off Command Procedures	6-7
6.2.5	SCI .S$NEWS File	6-8
6.3	SCI Language	6-8
6.3.1	Command Format	6-8
6.3.2	Special Characters	6-9
6.3.3	Variable Types	6-11
6.3.4	Field Prompts	6-11
6.3.4.1	ACNM Field Prompt Type	6-13
6.3.4.2	INT Field Prompt Type	6-13
6.3.4.3	NAME Field Prompt Type	6-14
6.3.4.4	STRING Field Prompt Type	6-14
6.3.4.5	YESNO Field Prompt Type	6-15
6.3.4.6	Abbreviating Field Prompts	6-15
6.3.4.7	The Dollar Sign in Initial Values	6-15
6.3.5	Synonyms	6-16
6.3.5.1	Types of Synonyms	6-16
6.3.5.2	Synonym Evaluation	6-17
6.3.6	Keywords	6-19
6.3.7	SCI Primitives	6-19
6.3.7.1	.PROC and .EOP Primitives	6-21
6.3.7.2	.BID, .QBID, .DBID, and .TBID Primitives	6-22
6.3.7.3	.DATA and .EOD Primitives	6-24
6.3.7.4	.EVAL Primitive	6-26
6.3.7.5	.EXIT Primitive	6-26
6.3.7.6	.IF, .ELSE, and .ENDIF Primitives	6-27
6.3.7.7	.LOOP, .UNTIL, .WHILE, and .REPEAT Primitives	6-28

Paragraph	Title	Page
6.3.7.8	.MENU Primitive	6-30
6.3.7.9	.OPTION Primitive	6-31
6.3.7.10	.OVLY Primitive	6-32
6.3.7.11	.PROMPT Primitive	6-33
6.3.7.12	.SHOW Primitive	6-34
6.3.7.13	.SPLIT Primitive	6-34
6.3.7.14	.STOP Primitive	6-36
6.3.7.15	.SYN Primitive	6-37
6.3.7.16	.USE Primitive	6-37
6.3.8	Processor Interfacing Subroutines	6-39
6.3.8.1	String Utility Subroutines	6-40
6.3.8.2	SCI Interface Subroutines	6-43
6.3.8.3	Arithmetic Utility Subroutines	6-52
6.3.8.4	Terminal Local File (TLF) Access Subroutines	6-53
6.4	SCI Environment and Batch Stream Operation	6-55
6.5	Examples	6-60
6.5.1	Command Procedure Examples	6-60
6.5.2	Command Processor Example	6-62
6.5.3	Batch Stream Listing	6-64
6.6	Error Messages	6-64
6.6.1	Unknown Volume Name	6-65
6.6.2	9001 — Invalid Access Name Syntax	6-65
6.6.3	9003 — Invalid Keyword Syntax	6-65
6.6.4	9005 — Invalid Command Name Syntax	6-65
6.6.5	9006 — Invalid Relation Name	6-65
6.6.6	9007 — Invalid Type Specification	6-65
6.6.7	900A — Spurious Characters at End	6-66
6.6.8	900E — Unknown Command Name	6-66
6.6.9	900F — Unknown Keyword	6-66
6.6.10	9011 — Required Argument Not Present	6-66
6.6.11	9019 — Invalid Keyword Value	6-66
6.6.12	FF02 — PROC Library Error	6-67
6.6.13	FF0B — Keyword Table Overflow	6-67

7 — Using Supervisor Calls (SVCs)

Paragraph	Title	Page
7.1	Introduction	7-1
7.2	Supervisor Call Definition	7-1
7.2.1	Coding a Supervisor Call	7-2
7.2.2	Defining Supervisor Call Blocks	7-3
7.2.3	Returning the Error Code	7-8

Paragraph	Title	Page

8 — Program Support Calls

8.1	General	8-1
8.2	Program Control SVCs	8-3
8.2.1	>02 — Time Delay SVC	8-3
8.2.2	>04 — End of Task SVC	8-3
8.2.3	>06 — Unconditional Wait SVC	8-3
8.2.4	>07 — Activate Suspended Task SVC	8-4
8.2.5	>09 — Do Not Suspend SVC	8-7
8.2.6	>0E — Activate Time Delay Task SVC	8-7
8.2.7	>11 — Change Priority SVC	8-8
8.2.8	>14 — Load an Overlay SVC	8-9
8.2.9	>16 — End of Program SVC	8-10
8.2.10	>17 — Get Parameters SVC	8-11
8.2.11	>1F — Scheduled Bid Task SVC	8-11
8.2.12	>2B — Execute Task SVC	8-13
8.2.13	>2E — Self-Identification SVC	8-15
8.2.14	>2F — End Action Status SVC	8-15
8.2.15	>31 — Map Program Name to ID SVC	8-16
8.2.16	>35 — Poll Status of Task SVC	8-18
8.2.17	>3E — Reset End Action SVC	8-20
8.3	Memory Control	8-21
8.3.1	>10 — Get Common Data Address SVC	8-21
8.3.2	>12 — Get Memory SVC	8-22
8.3.3	>13 — Release Memory SVC	8-23
8.3.4	>1B — Return Common Data Address SVC	8-24
8.4	Intertask Communications	8-24
8.4.1	>1C — Putdata SVC	8-24
8.4.2	>1D — Getdata SVC	8-25
8.5	Data Conversion Services	8-27
8.5.1	>0A — Convert Binary-to-Decimal SVC	8-27
8.5.2	>0B — Convert Decimal-to-Binary SVC	8-28
8.5.3	>0C — Convert Binary-to-Hexadecimal SVC	8-29
8.5.4	>0D — Convert Hexadecimal-to-Binary SVC	8-30
8.6	System Information	8-31
8.6.1	>03 — Date and Time SVC	8-31
8.6.2	>21 — System Log SVC	8-32
8.6.3	>3F — Retrieve System Information SVC	8-33

9 — Device I/O Supervisor Calls

9.1	Device Independent I/O	9-1
9.2	Device Dependent I/O	9-2
9.3	Programming Considerations	9-3
9.4	Device I/O Call Blocks	9-4
9.5	Device Dependent/Independent I/O SVCs	9-15
9.5.1	>00 Subopcode — Open SVC	9-16
9.5.1.1	Data Terminals	9-16

Paragraph	Title	Page
9.5.1.2	Printer Devices	9-16
9.5.1.3	Video Display Terminals	9-16
9.5.2	>01 Subopcode — Close SVCs	9-17
9.5.3	>02 Subopcode — Close with EOF SVC	9-17
9.5.4	>03 Subopcode — Open Rewind SVC	9-17
9.5.5	>04 Subopcode — Close Unload SVC	9-17
9.5.6	>05 Subopcode — Read Device Status SVC	9-17
9.5.6.1	Magnetic Tape	9-17
9.5.6.2	Video Display Terminals	9-17
9.5.6.3	Disk Devices	9-19
9.5.6.4	Teleprinter Devices	9-20
9.5.6.5	Line Printers	9-22
9.5.7	>06 Subopcode — Forward Space SVC	9-24
9.5.8	>07 Subopcode — Backward Space SVC	9-24
9.5.9	>09 Subopcode — Read ASCII SVC	9-24
9.5.9.1	Magnetic Tape Unit	9-24
9.5.9.2	Video Display Terminal	9-24
9.5.9.3	Teleprinter Devices	9-24
9.5.9.4	733 ASR Cassette Unit	9-25
9.5.9.5	Card Reader	9-25
9.5.9.6	Other Devices	9-25
9.5.10	>0A Subopcode — Read Direct SVC	9-25
9.5.10.1	733 ASR Cassette Unit	9-25
9.5.10.2	Magnetic Tape Unit	9-25
9.5.10.3	Card Reader	9-25
9.5.10.4	Video Display Terminal	9-25
9.5.10.5	820 KSR	9-25
9.5.10.6	Teleprinter Devices	9-26
9.5.10.7	Other Devices	9-26
9.5.11	>0B Subopcode — Write ASCII SVC	9-26
9.5.11.1	Magnetic Tape	9-26
9.5.11.2	Keyboard/Printer	9-26
9.5.11.3	Line Printer	9-26
9.5.11.4	Video Display Terminal	9-26
9.5.11.5	733 ASR Cassette Unit	9-26
9.5.11.6	820 KSR	9-26
9.5.11.7	Teleprinter Devices	9-27
9.5.11.8	Other Devices	9-27
9.5.12	>0C Subopcode — Write Direct SVC	9-27
9.5.12.1	733 ASR Cassette Unit	9-27
9.5.12.2	Magnetic Tape Unit	9-27
9.5.12.3	Video Display Terminal	9-27
9.5.12.4	820 KSR	9-27
9.5.12.5	Teleprinter Devices	9-27
9.5.12.6	Line Printer	9-27
9.5.12.7	Other Devices	9-27
9.5.13	>0D Subopcode — Write EOF SVC	9-28
9.5.13.1	733 ASR Cassette Unit	9-28
9.5.13.2	Keyboard/Printer	9-28

Paragraph	Title	Page
9.5.13.3	Line Printer	9-28
9.5.13.4	Video Display Terminal	9-28
9.5.13.5	Card Reader	9-28
9.5.13.6	Magnetic Tape Unit	9-28
9.5.13.7	820 KSR	9-28
9.5.13.8	Teleprinter Devices	9-28
9.5.14	>0E Subopcode — Rewind SVC	9-28
9.5.15	>0F Subopcode — Unload SVC	9-28
9.5.16	>91 Subopcode — Assign LUNO SVC	9-28
9.5.17	>93 Subopcode — Release LUNO SVC	9-29
9.5.18	>99 Subopcode — Verify Device Name SVC	9-29
9.6	Key Types	9-29
9.6.1	Data Keys	9-29
9.6.2	Event Keys	9-29
9.6.3	Task Edit Keys	9-31
9.6.4	System Edit Keys	9-32
9.6.5	Repeat Character Compression	9-33
9.6.6	Character Validation	9-34
9.6.7	Hard Break Key Sequence	9-36
9.7	Other I/O Related Calls	9-36
9.7.1	>01 — Wait for I/O SVC	9-36
9.7.2	>30 — Get Event Key by ID SVC	9-37
9.7.3	>36 — Wait on Multiple Initiate I/O SVC	9-37
9.7.4	>39 — Get Event Key by LUNO SVC	9-38
9.7.5	>0F — Abort I/O on Specified LUNO SVC	9-38
9.8	Pass Thru Mode	9-38
9.9	Edit Flag Words	9-39

10 — File I/O SVCs

Paragraph	Title	Page
10.1	Introduction	10-1
10.2	File I/O SVCs	10-1
10.2.1	Sequential and Relative Record File I/O Call Block	10-1
10.2.2	Sequential and Relative Record File Operations	10-5
10.2.2.1	>00 Subopcode — Open SVC	10-5
10.2.2.2	>01 Subopcode — Close SVC	10-5
10.2.2.3	>02 Subopcode — Close EOF SVC	10-7
10.2.2.4	>03 Subopcode — Open Rewind SVC	10-7
10.2.2.5	>04 Subopcode — Close Unload SVC	10-7
10.2.2.6	>05 Subopcode — Read File Characteristics SVC	10-7
10.2.2.7	>06 Subopcode — Forward Space SVC	10-9
10.2.2.8	>07 Subopcode — Backward Space SVC	10-9
10.2.2.9	>09 Subopcode — Read ASCII SVC	10-9
10.2.2.10	>0A Subopcode — Read Direct SVC	10-9
10.2.2.11	>0B Subopcode — Write ASCII SVC	10-9
10.2.2.12	>0C Subopcode — Write Direct SVC	10-9
10.2.2.13	>0D Subopcode — Write EOF SVC	10-10
10.2.2.14	>0E Subopcode — Rewind SVC	10-10

Paragraph	Title	Page
10.2.2.15	>0F Subopcode — Unload SVC	10-10
10.2.2.16	>10 Subopcode — Rewrite SVC	10-10
10.2.2.17	>11 Subopcode — Modify Access Privileges SVC	10-10
10.2.2.18	>12 Subopcode — Open Extend SVC	10-10
10.2.2.19	>4A Subopcode — Unlock SVC	10-10
10.2.3	Key Indexed Files	10-10
10.2.3.1	Key Indexed File Keys	10-11
10.2.3.2	Key Indexed File Records	10-11
10.2.3.3	Key Indexed File Key and Record Example	10-12
10.2.3.4	Key Indexed File Algorithm	10-12
10.2.4	Key Indexed File Call Block and Currency Blocks	10-13
10.2.5	Key Indexed File SVC Subopcodes	10-14
10.2.5.1	>00 Subopcode — Open SVC	10-15
10.2.5.2	>01 Subopcode — Close SVC	10-16
10.2.5.3	>03 Subopcode — Open Rewind SVC	10-16
10.2.5.4	>05 Subopcode — Read File Characteristics SVC	10-17
10.2.5.5	>06 Subopcode — Forward Space SVC	10-18
10.2.5.6	>07 Subopcode — Backward Space SVC	10-19
10.2.5.7	>09 Subopcode — Read ASCII SVC	10-19
10.2.5.8	>0A Subopcode — Read Direct SVC	10-19
10.2.5.9	>0E Subopcode — Rewind SVC	10-20
10.2.5.10	>40 Subopcode — Open Random SVC	10-20
10.2.5.11	>41 Subopcode — Read Greater SVC	10-21
10.2.5.12	>42 Subopcode — Read by Key/Read Current SVC	10-21
10.2.5.13	>44 Subopcode — Read Greater or Equal	10-22
10.2.5.14	>45 Subopcode — Read Next SVC	10-23
10.2.5.15	>46 Subopcode — Insert SVC	10-24
10.2.5.16	>47 Subopcode — Rewrite SVC	10-24
10.2.5.17	>48 Subopcode — Read Previous SVC	10-25
10.2.5.18	>49 Subopcode — Delete by Key/Delete Current SVC	10-26
10.2.5.19	>4A Subopcode — Unlock Current SVC	10-27
10.2.5.20	>50 Subopcode — Set Currency Equal SVC	10-28
10.2.5.21	>51 Subopcode — Set Currency Greater or Equal	10-28
10.2.5.22	>52 Subopcode — Set Currency Greater	10-28
10.2.5.23	Using Partial Keys	10-29
10.2.5.24	Error and Informative Codes	10-29
10.2.5.25	Estimating Key Indexed File Size	10-31
10.3	File Utility SVCs	10-36
10.3.1	>90 Subopcode — Create File SVC	10-41
10.3.1.1	Create Sequential File Example	10-44
10.3.1.2	Create Key Indexed File Example	10-45
10.3.2	>91 Subopcode — Assign LUNO SVC	10-46
10.3.3	>92 Subopcode — Delete File SVC	10-47
10.3.4	>93 Subopcode — Release LUNO SVC	10-47
10.3.5	>95 Subopcode — Rename File SVC	10-48
10.3.6	>96 Subopcode — Unprotect File SVC	10-48
10.3.7	>97 Subopcode — Write Protect File SVC	10-48
10.3.8	>98 Subopcode — Delete Protect File SVC	10-48
10.3.9	>99 Subopcode — Verify Pathname SVC	10-48

Paragraph	Title	Page
10.3.10	>9A Subopcode — Add Alias SVC	10-49
10.3.11	>9B Subopcode — Delete Alias SVC	10-49
10.3.12	>9C Subopcode — Define Write Mode SVC	10-49
10.4	Temporary Files	10-50

11 — Debugging a Program

Paragraph	Title	Page
11.1	General Information	11-1
11.2	Modes of Debugging	11-1
11.2.1	Unconditional Suspend	11-2
11.2.2	Symbols	11-3
11.2.3	Expressions	11-4
11.3	Commands for All Tasks	11-6
11.3.1	Data Display Commands	11-8
11.3.1.1	List Breakpoints — LB	11-8
11.3.1.2	List Logical Record — LLR	11-9
11.3.1.3	List Memory — LM	11-9
11.3.1.4	List System Memory — LSM	11-10
11.3.1.5	Show Absolute Disk — SAD	11-11
11.3.1.6	Show Allocatable Disk Unit — SADU	11-12
11.3.1.7	Show Internal Registers — SIR	11-13
11.3.1.8	Show Panel — SP	11-14
11.3.1.9	Show Program Image — SPI	11-14
11.3.1.10	Show Relative to File — SRF	11-15
11.3.1.11	Show Value — SV	11-16A
11.3.1.12	Show Workspace Registers — SWR	11-16A
11.3.2	Data Modification Commands	11-16A
11.3.2.1	Modify Absolute Disk — MAD	11-16B
11.3.2.2	Modify Allocatable Disk Unit — MADU	11-18A
11.3.2.3	Modify Internal Registers — MIR	11-18B
11.3.2.4	Modify Memory — MM	11-19
11.3.2.5	Modify Program Image — MPI	11-19
11.3.2.6	Modify Relative to File — MRF	11-20
11.3.2.7	Modify System Memory— MSM	11-22
11.3.2.8	Modify Workspace Registers — MWR	11-22
11.3.3	Breakpoint Commands	11-23
11.3.3.1	Assign Breakpoints — AB	11-23
11.3.3.2	Delete Breakpoints — DB	11-24
11.3.3.3	Delete and Proceed from Breakpoint — DPB	11-24
11.3.3.4	Proceed from Breakpoint — PB	11-25
11.3.4	Task Control Commands	11-25
11.3.4.1	Activate Task — AT	11-25
11.3.4.2	Halt Task — HT	11-26
11.3.4.3	Resume Task — RT	11-27
11.3.4.4	Execute in Debug Mode — XD	11-27
11.3.4.5	Execute and Halt Task — XHT	11-28
11.3.5	Search Commands	11-29
11.3.5.1	Find Byte — FB	11-29

Paragraph	Title	Page
11.3.5.2	Find Word — FW	11-29
11.3.6	Controlled Task Commands	11-30
11.3.6.1	Assign Simulated Breakpoint — ASB	11-30
11.3.6.2	Delete Simulated Breakpoints — DSB	11-32
11.3.6.3	List Simulated Breakpoints — LSB	11-32
11.3.6.4	Quit Debug Mode — QD	11-32
11.3.6.5	Resume Simulated Task — RST	11-33
11.3.6.6	Simulate Task — ST	11-33
11.4	Station Dependent Displays	11-35

12 — International Considerations of DX10

12.1	Introduction	12-1
12.2	Country Code	12-2
12.3	Information Interchange Codes	12-3
12.4	Optional SCI Prompt	12-3
12.5	Key Indexed File Collating Sequences	12-3
12.6	International Devices	12-4
12.7	IPF — International Print File Command	12-6
12.7.1	IPF Command Format	12-6
12.7.2	IPF Command User Responses	12-7
12.7.3	IPF Command Example	12-8
12.7.4	IPF Error Messages	12-9
12.8	SCC — Show Country Code Command	12-10
12.8.1	Command Format	12-10
12.8.2	SCC Command User Responses	12-10
12.8.3	SCC Command Example	12-10

Appendixes

Appendix	Title	Page
A	Keycap Cross-Reference	A-1
B	ASCII Device I/O Operations	B-1
C	Task State Codes	C-1
D	Reentrant Programming Example in Assembly Language	D-1
E	File and Device I/O SVC Call Blocks	E-1
F	SVC Codes	F-1

Index

Illustrations

Figure	Title	Page
2-1	Mapping	2-3
2-2	Tasks Sharing Segments	2-4
2-3	Task Memory Configuration	2-6
3-1	I/O Process Paths	3-26
6-1	SF Command Procedure Example	6-4
7-1	XOP Call Processing	7-2
8-1	Task Synchronization	8-5
9-1	Device I/O SVC Call Block for Assign	9-5
9-2	Device I/O SVC Call Block with Extended Block	9-8
10-1	Sequential and Relative Record File I/O Call Block	10-2
10-2	KIF Call Block, Currency Block, and KEY Relationship	10-13
10-3	File Utility SVC Call Block	10-37
10-4	Key Indexed File Definition Call Block	10-43
A-1	911 VDT Standard Keyboard Layout	A-9
A-2	915 VDT Standard Keyboard Layout	A-10
A-3	940 EVT Standard Keyboard Layout	A-11
A-4	931 VDT Standard Keyboard Layout	A-12
A-5	Business System Terminal Standard Keyboard Layout	A-13
A-6	820 KSR Standard Keyboard Layout	A-14
B-1	911 VDT Graphics Character Keyboard Positions	B-7
B-2	931 VDT Graphics Characters	B-12
B-3	940 EVT Graphics Characters	B-16

Tables

Table	Title	Page
3-1	File Usage Tradeoffs	3-4
3-2	File Access Mode Compatibility	3-10
3-3	Format Information for Supported Disks	3-14
3-4	Blocking Logical Records for Sequential Files	3-21
6-1	SCI Special Characters	6-10
6-2	Valid Field Prompt Types	6-11
6-3	Standard Synonyms	6-17
6-4	SCI Primitives	6-20

Table	Title	Page
6-5	Command Privilege Levels	6-21
8-1	Program Support Supervisor Calls	8-2
9-1	Device I/O SVCs and Call Block Requirements	9-3
9-2	Device I/O Assign and Release LUNO Call Block Bit Assignments	9-6
9-3	General/Extended I/O SVC Call Block Bit Assignments	9-9
9-4	Device Dependent Responses to I/O Subopcodes	9-15
9-5	Returned Device Types and Default Record Lengths	9-16
10-1	File I/O SVC Call Block Bit Assignments	10-3
10-2	Information Returned for the Read File Characteristics Operation	10-8
10-3	Information Returned for Read File Characteristics Operation	10-17
10-4	Error and Informative Codes for Key Indexed File Subopcodes	10-30
10-5	File Utility Call Block Bit Assignments	10-38
10-6	Key Indexed File Definition Bit Assignments	10-42
11-1	Debug Commands	11-7
11-2	Command Displays	11-35
12-1	DSR Codes for International Characters	12-5
A-1	Generic Keycap Names	A-2
A-2	Frequently Used Key Sequences	A-8
A-3	911 Keycap Name Equivalents	A-8
B-1	911 VDT Key Character Code Transformations	B-3
B-2	931 VDT Key Character Code Transformations	B-8
B-3	940 EVT Key Character Code Transformations	B-13
B-4	Business System Terminal Key Character Code Transformations	B-17
B-5	820 KSR Key Character Code Transformations	B-21
B-6	783 TPD Key Character Code Transformations	B-24
B-7	Display Terminal Graphics Character Sets	B-27
B-8	733 and 743 Terminal Character Set	B-29
B-9	Card Reader Character Set	B-32
B-10	Line Printer Character Set	B-34
C-1	Task State Codes	C-1

1

Introduction

1.1 GENERAL

The *DX10 Applications Programming Guide* (Volume III) provides two specific types of information:

- General information about DX10 facilities as they relate to applications programming

- How DX10 supports applications programming in assembly language

The general information focuses on DX10 program management and I/O system support. This information is important to every programmer of DX10, regardless of the programming language used. The DX10 operating system supports both assembly language and high-level languages as discussed in the following paragraphs.

The high-level languages supported on DX10 are COBOL, FORTRAN, BASIC, Pascal, and RPG II. Each high-level language has unique features, and supports DX10 facilities a little differently. Information about specific features of a high-level language is located in the language documentation for the particular language. Each language has an associated reference manual and programming guide. This manual (Volume III) bridges the gap between the respective language manuals and specific DX10 operating system information by providing background information about DX10.

The 990/99000 assembly language is supported by an assembly language reference manual, and this manual, which functions as an assembly language programming guide.

The official titles and part numbers of all the language documents are furnished in the frontispiece. Before reading this manual, you should be familiar with the DX10 operating system as it is described in *DX10 Operating System Concepts and Facilities* (Volume I).

1.2 APPLICATIONS PROGRAMMING ON DX10

DX10 provides full facilities for developing complete applications. In a general sense, DX10 allows you to develop application programs without having to write I/O routines or file management routines. DX10, along with the languages it supports, provides the processing support associated with an operating system.

In high-level languages, you issue available statements from the specific language to request operating system services. Facilities not directly supported by DX10 are supported by the language run-time packages.

Services are requested in assembly language by issuing supervisor calls (SVCs) to perform the operation. You must code the SVCs in the program using SVC call blocks. SVC call blocks contain the parameters required for each predefined SVC. SVCs are available to perform such program management and I/O services as executing tasks, opening and closing files, and so on. Information on how to use SVCs in an assembly language program can be found in Section 7 of this manual. Each available SVC and associated call block is documented according to its general function in Sections 8, 9, and 10.

Developing complete applications includes writing command procedures using the System Command Interpreter (SCI) programming language. User command procedures supply the operator/program interface by displaying prompts on the screen so that the operator can submit information required by the program. These command procedures also function as master programs by calling the application programs to execute. Information on the SCI programming language and on writing user-friendly command procedures for your applications is located in Section 6.

For a complete overview of the six-volume DX10 document set, refer to Volume I. Volume I contains a section dedicated to the documentation organization, and also contains a master subject index to help you find the information you need.

2

How DX10 Manages Programs

2.1 INTRODUCTION

A program is a collection of machine instructions which directs the activities of a computer. Under DX10, any activation of a program is called a task. There can be several activations of the same program at a given time but each activation is a different task. For example, the System Command Interpreter (SCI) is a program and each station may have, as a task, a unique activation of the SCI program. A program becomes a task when DX10 assigns a run-time ID. A program may become several identical tasks if that program is activated at several different stations or by other tasks. DX10 assigns different run-time IDs to each activation, making each activation a unique task.

When you install an application program on a program file, you assign it a hexadecimal ID number which becomes the installed ID. Each time that program is executed, DX10 assigns it a run-time ID. All programs that are part of the DX10 operating system are already installed on the system program file, and have unique installed IDs. The Show Task Status (STS) SCI command lists installed IDs and run-time IDs for all tasks that are active in the system at any given time. (Internal system tasks such as file handling routines and so on are excluded.)

2.2 PROGRAM STRUCTURE

DX10 allows you to structure programs in several ways. Program structure affects the program's efficiency, and different structures enhance different applications. There are three components of program structure, as follows:

- Procedure segments
- Task segments
- Overlay segments

Procedure segments are usually reentrant. When writing a program, you can implement code that can also be used by other programs, or that can be used repeatedly by the same program without reinitializing any of the variables. This is called reentrant code because it can be entered repeatedly at the same address by any task and still execute properly. (Reentrancy is discussed in detail later in this section.) If a segment of code is reentrant, other programs can use it at the same time. If you want a segment of code to be sharable, install it on a program file as a procedure segment. The technique for installing procedure segments is discussed later.

The task segment is the part of the program that is unique; it can only be executed one time and then must be reinitialized. For example, code containing data elements that are increased, decreased or changed by the program during execution, cannot be reused by the program. This type of code is nonreentrant and it cannot be shared. When you install a segment of code as a task, it is understood to be nonsharable. While not all programs need a procedure segment, all programs need a task segment. This is explained more fully in the paragraphs on program segmentation.

Overlay segments are a third type of structure you can use. Overlays save memory space by replacing seldom-used code in a task segment. This reduces the memory required for the task segment.

DX10 mapping techniques control program segmentation structure. Program segments and mapping are discussed in the following paragraphs.

2.2.1 Program Segmentation

You can segment programs into one of the following configurations:

- A single segment, including executable code and any required data. Install this configuration on a program file as a task segment. Each instance of the program in execution is a separate task.
- Two separately loadable segments consisting of a procedure segment and a task segment. The procedure segment contains reentrant executable code. The task segment can contain executable code (usually nonreentrant) and/or local data.
- Three separately loadable segments consisting of two procedure segments and a task segment.

NOTE

Programs with fewer than three segments can access system common. Refer to the discussion of the Get Common SVC.

You can install task, procedure, and overlay segments on the program file using SCI commands. These commands are the Install Task (IT) SCI command for task segments, the Install Procedure (IP) SCI command for procedure segments, and the Install Overlay (IO) SCI command for overlay segments. You can also install tasks, procedures, and overlays using the Link Editor, as described in the *Link Editor Reference Manual*. Privileged SVCs that install tasks, procedures and overlays from another task are described in the *DX10 System Programming Guide* (Volume V).

Program segmentation, linking and installing segments for programs written in high-level languages are discussed in the individual language manual, but understanding how DX10 handles segments is important. If you are programming in assembly language, the specifics of program segmentation are very important, since you must explicitly create the elements of a program to allow you to link them together and install them with the desired characteristics.

2.2.2 Program Mapping

The computer hardware supported by the DX10 operating system uses a 20-bit memory address bus and can address 2048K bytes of memory. The logical address space available to a task is limited by a 16-bit word address to 64K bytes. The difference is resolved by the mapping hardware in the computer that maps the logical address space onto the computer's physical memory. This mapping hardware allows the operating system to access one, two, or three segments of memory at the same time, as if they were contiguous segments. The maximum logical size of any task is 65,504 bytes. This size represents 64K bytes less 32 bytes. Figure 2-1 illustrates segment mapping.

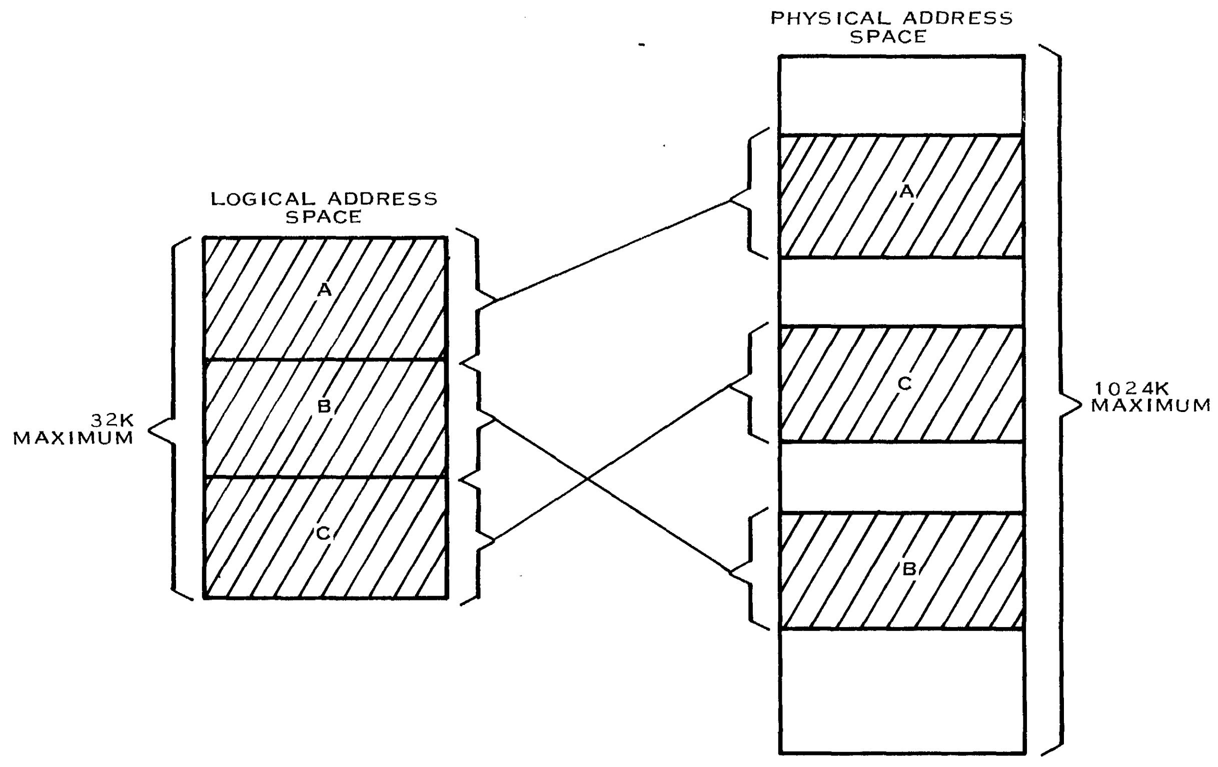


Figure 2-1. Mapping

Segments in the physical address space need not be contiguous. Since DX10 maintains separate mapping parameters for each task, each task may consist of one, two or three segments with a total extent of 65,504K bytes. Several tasks may share one or two segments. One segment, the task segment, is unique to each instance of a program. The shareable segments of a task are procedure segments. Figure 2-2 illustrates two tasks sharing two segments of memory. The two tasks could be instances of the same program. For example, both tasks might be instances of the SCI program executing at different stations.

The computer instructions that control mapping are reserved for system use. Use of these instructions by nonprivileged user tasks causes fatal errors. DX10 memory management controls mapping so that the mapping function is transparent to the execution of user tasks.

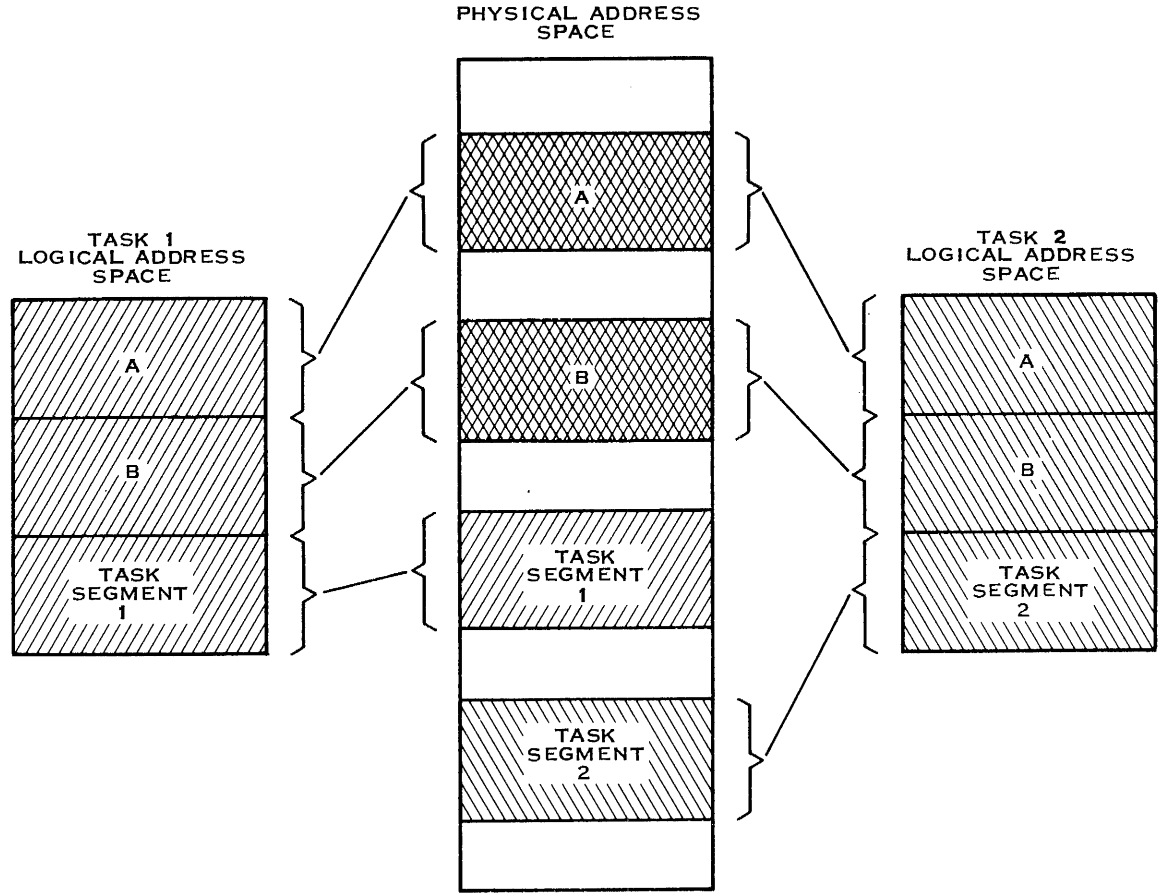


Figure 2-2. Tasks Sharing Segments

Since DX10 manages memory in 32-byte blocks, the following boundary rules apply for programs consisting of two or three separate segments:

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

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

2.2.2.1 Implementing Program Segmentation. Figure 2-3 illustrates the possible memory configurations for programs under DX10. The following paragraphs outline the processes for achieving these configurations, primarily from an assembly language programming point of view. The maximum allowable memory for each task in any configuration is 65,504 bytes.

2.2.2.2 Task Segment Only. Figure 2-3 illustrates five configurations, denoted with letters A through E. Illustration A shows a program with a task segment only. The following process generates this configuration:

1. Assemble (or compile) the source program. This can be done in several modules.
2. Link the object code into one module using the Link Editor if more than one object module is involved, or if a language runtime must be linked in.
3. Install the linked object module on a program file as a task segment using the Install Task (IT) SCI command, or as a function of the Link Editor using IMAGE format.
4. Execute the installed task using an Execute Task (XT) SCI command, an Execute Task SVC, or the .BID SCI command. (Refer to paragraph 2.3 for alternative ways to execute a program.)

NOTE

Some high-level languages must have their runtime linked as the first procedure. Therefore they cannot be linked as a task only. Refer to the applicable language programmer's guide.

2.2.2.3 Task Segment and System Common Segment. In Figure 2-3, Illustration B shows the memory configuration of a program with a task segment and a system common segment. The process generating this configuration is the same process generating a task-only configuration, except that the task itself must issue a Get Common SVC during execution. Executing a Release Common SVC releases the common segment, but it need not be executed before program termination. (The system common area must be defined during system generation.)

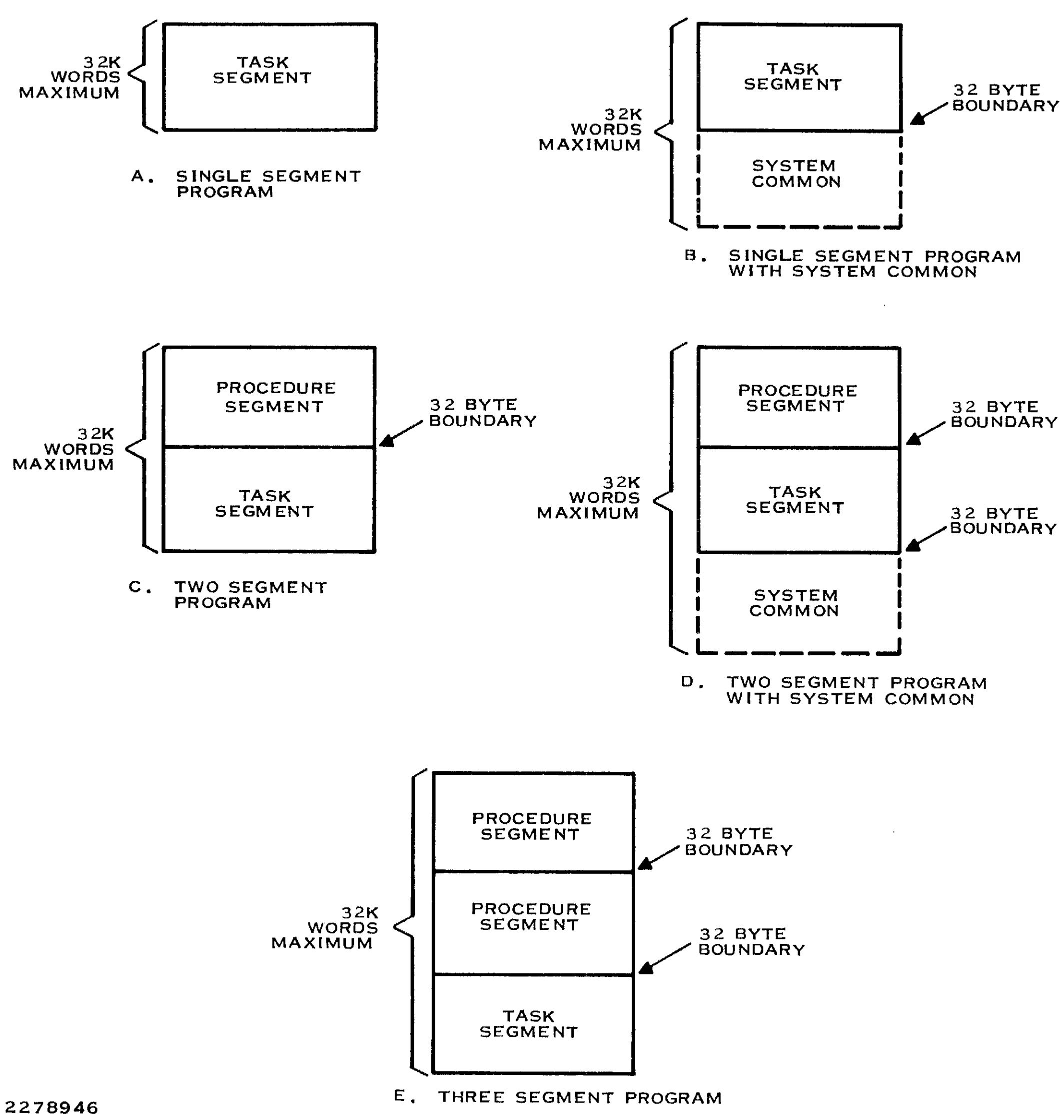


Figure 2-3. Task Memory Configuration

2.2.2.4 Task Segment and One or Two Procedure Segments. Illustration C and E in Figure 2-3 show a program with a task segment and procedure segment(s). The following process generates this configuration:

1. Assemble the various modules of the program separately.

2. Using the Link Editor, link the two (or three) segments of the program. You must use the PROCEDURE Link Editor command when linking to specify the separate procedures. Otherwise, they will not be separately loadable.

3. Install the procedure(s) using the Install Procedure (IP) SCI command (or by using the Link Editor IMAGE format).

4. Install the task segment using the IT SCI command and using the procedure IDs specified during the IP command (or by using the Link Editor).

5. Execute the installed task using an XT command, an Execute Task SVC, or the .BID SCI command. (Refer to paragraph 2.3 for alternative ways of executing a program.)

2.2.2.5 Task Segment, Procedure Segment, and System Common. Illustration D of Figure 2-3 shows a program with two segments and accessing system common. The process generating this configuration is the same process generating the program with a task segment and one procedure segment configuration except that the task itself must issue a Get Common SVC during execution. Programs consisting of a task segment and two procedure segments may not access system common, since system common is treated as a segment, and only three segments are allowed.

2.2.2.6 Overlays. Overlays are unshared parts of the task segment. Procedures cannot have overlayed code. You define and generate overlay structures using the Link Editor for assembly language programs. High-level languages can use other methods for implementing overlays. Refer to the *Link Editor Reference Manual* or the appropriate language programmer's guide for more information on organizing overlay structures. How DX10 handles overlays is discussed later in this section. Refer to Section 8 for a description of the Load Overlay SVC.

2.3 TASK EXECUTION

Task execution is initiated through the following SVCs:

- Execute Task SVC (code >2B)
- Bid Task SVC (code >05)
- Scheduled Bid Task SVC (code >1F)

The Execute Task SVC is the most common method. The Bid Task SVC is included only for compatibility with pre-3.X versions of DX10. Scheduled Bid Task executes at a prescheduled time.

There are four ways to access the Execute Task SVC:

- Execute the XT, XHT, or XTS SCI command
- Code the SVC call within a program
- Execute the .BID, .QBID, .DBID, or .TBID primitives of the SCI procedure language, or call the SCI routine S$BIDT
- Execute a language runtime routine provided for that purpose

If the executing program needs access to synonyms and parameters in the terminal communications area (TCA) as set up by SCI, use .BID, .QBID, .DBID, or .TBID primitives from an SCI command procedure, or the S$BIDT interface routine to execute the program. Only these avenues save the synonyms and parameters. If the executing program does not need that access, you can use any of the available avenues. S$BIDT can only be used in a task that was itself bid using one of the avenues that preserves synonyms and parameters.

2.4 HOW DX10 MANAGES TASK EXECUTION

Task execution management is based on task scheduling. Task scheduling refers to the way the system chooses a task to receive control of the CPU. Scheduling also encompasses how user memory is managed, since a task must be in memory in order to execute, and must be loaded into memory for that purpose. Sometimes other tasks must relinquish the space they occupy so tasks scheduled for execution can be loaded into that memory space.

2.4.1 Task Scheduling

DX10 always allocates the CPU to the highest priority task awaiting execution. However, certain conditions can cause a task's priority to be changed if the Task Sentry is enabled. (Task Sentry is discussed later in this manual.)

DX10 schedules or reschedules tasks awaiting execution when one of the following occurs:

- An external interrupt bids a task
- An event completes for which a task is waiting
- The executing task suspends
- Task sentry lowers the priority of the executing task (if Task Sentry is enabled as discussed later in this section)
- The time slice allocated to a task expires (if time slicing is enabled as discussed later in this section)
- A time delayed task is due to become active

An external interrupt bids a task in response to an event external to the CPU. Typing in the character sequence requesting log-on is an example of such an event. The interrupt signals the task scheduler that a specific user program needs to be scheduled to run. The task scheduler examines the queue of tasks awaiting execution, which now includes the newly-bid task, and selects the task of highest priority for execution.

Rescheduling occurs when an event for which a task is waiting completes. This is necessary because the suspended task reactivates and requires CPU time. Completion of disk I/O for a task is an example.

If the executing task suspends, the task scheduler allocates control of the CPU to the highest priority task in the system that is ready to execute. A task is ready to execute if it is not waiting on any other events, such as I/O.

If Task Sentry is enabled and lowers the priority of the currently executing task, the scheduler reschedules the tasks awaiting execution. This ensures that the task with highest priority executes.

If time slicing is enabled, tasks execute for a fixed interval of time. Upon expiration of a time slice, a rescheduling of the tasks is performed. Time slicing allows tasks of equal priority to share the CPU in a round-robin fashion.

If a time delayed task becomes active, a rescheduling of the tasks in the system is performed to ensure that the highest priority task is executing.

NOTE

Task Sentry, time slicing, and time slice duration are options selected during system generation. System generation is discussed in the Volume V.

2.4.1.1 Priority. The DX10 operating system requires that each task be assigned a priority level when it is installed. This priority level is one of the major determining factors in how the task scheduler schedules and reschedules tasks awaiting execution.

DX10 provides the following 132 levels of priority:

	Level	Meaning
(Highest)	0	Critical system tasks (reserved for DX10)
	R1-R127	Real-time priorities
	1	Foreground interactive tasks
	2	Foreground compute bound tasks
(Lowest)	3	Background tasks
(Floating)	4	(Floats between priority 1, 2 in foreground, if background, runs at 3)

Priority *level 0* is intended for the most critical system functions and *is reserved for DX10 internal use only.* The remainder of system tasks are distributed appropriately among the lower priority levels with regard to their relative importance.

Real-time priorities provide the user the capability to supersede all except the most important system tasks. For applications requiring prompt access of the CPU, DX10 delays some lower priority system functions in an effort to schedule real-time tasks.

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

For programs requiring user interaction and multiple disk accesses, priority level 4 automatically switches between priority levels 1 and 2 as the program executes. When the task requests input from a terminal it is assigned priority 1 so that when the user responds, the program can react quickly to the user's request. At other times, the task is assigned priority 2 so as not to interfere with other programs requesting input from a terminal.

Priority level 3 is for programs executing in background, and require no user interaction.

2.4.1.2 Task Sentry. An inherent characteristic of the DX10 task scheduling scheme is that tasks which are CPU-bound may lock out all tasks of a lower priority level. Task sentry monitors CPU-bound tasks and lowers their priority by one after a specified number of 50-millisecond intervals have elapsed. This allows lower priority tasks to be allocated CPU time. When a CPU-bound task suspends itself (for example, an I/O request), the task sentry resets the suspended task's priority to its installed priority level. In the case of floating priority tasks, their priority is reset to either priority level one or two. You enable the task sentry option during system generation, as described in Volume V.

Task sentry enables you to gain control of a runaway task. However, if such a runaway task has an SVC call in its loop, the priority is continually reset, preventing task sentry from lowering the task's priority. An initial program load (IPL) may be the only way to recover from such a situation.

2.4.1.3 How Priority Scheduling Affects Applications. Priority scheduling can affect applications because the scheduling algorithm always executes the highest priority task first. A task of a given priority can effectively prevent tasks of lesser priority from executing if the higher priority task requires a long execution time, and suspends for only very brief periods.

For example, any task with real-time priority that remains in execution, either through a runaway situation, or by simply requiring a long compute time, can effectively lock out any program of lesser priority. For tasks executing at priority level 2, a task with priority 1 can lock it out, and so on. Priority scheduling can also lock out SCI, since it operates at priority 1 or 2 (floating priority), and real-time priority takes precedence. Communication between tasks can be affected by such a situation.

A lower priority task that must communicate with another task within a specific period of time can be prevented from communicating if it is effectively locked out by a higher priority task. If the communication does not occur within the specific period, the task targeted for such communication determines that the lower priority task has terminated abnormally, and an error can result. This situation can occur with tasks that must communicate with Sort/Merge and TIFORM. An extra load on the system can cause the execution of any task of priority 3 to be delayed, since most other tasks execute at priority 1 or 2.

2.4.2 Dynamic Memory Management

DX10 manages memory by allocating memory to high priority tasks and rolling out low priority tasks. When a task already in memory is selected for execution, a check is made to see if a higher priority task is waiting to be loaded into memory. If a higher priority task is waiting, an attempt is made to load it. Allocating memory for the higher priority task can cause the low priority task to be rolled out of memory.

To obtain memory for the current task, DX10 copies (or rolls out) tasks occupying memory to the roll file which is a system file on disk. The following are eligible for roll-out:

- File blocking buffers (written to appropriate file)
- Disk-resident tasks
- Disk-resident procedures (after all attached tasks are rolled)

Once rolled out, a task remains on the roll file until it is selected for a time slice and brought back into memory (rolled in). Only the memory-resident file blocking buffer, memory-resident tasks and procedures are exempt from roll-out. Memory-resident tasks and procedures are those that are installed on the system program file with the corresponding attribute set.

2.5 PROGRAMMING CONSIDERATIONS

When you begin designing an application, you may wish to use some of the features that DX10 provides. For example, you may want to share code or data between tasks, or to use overlays to save memory. The following paragraphs discuss the way DX10 handles programs from the standpoint of what you need to know to produce an effective applications program structure.

2.5.1 Reentrancy and Sharing Code

A program is *reentrant* if it can be shared by several tasks. These tasks can be requested by several users at different terminals. The term *reentrant* comes from the ability of one copy of the code in memory to be reentered at the same address by several different tasks at different times, without reinitializing any of the variables. You may want to use reentrant code for various purposes. The primary use of reentrant code is to reduce the amount of computer memory that several programs may require by letting the programs share one copy of the common parts.

In DX10, you make a segment of code sharable by linking and installing it on a program file as a procedure segment. There are basically three types of sharable code, as outlined in the following list. Each type is discussed in the paragraphs following the list.

- Address-independent code, called truly reentrant
- Address-dependent code, called pseudo-reentrant because you must carefully arrange the data in the associated task segment to make it reentrant
- Data that gets modified during task execution, called a *dirty* procedure. If you want to share this data among several tasks, you can achieve a simulated reentrancy by carefully controlling each task's access to the data.

When programming in a high-level language such as COBOL, FORTRAN, or Pascal, you usually use only address-dependent code segments.

2.5.1.1 Address-Independent Reentrant Procedures. An address-independent reentrant procedure is one in which no routine linked into it modifies itself or any other routine or has any direct reference within itself to any code or data in the task segment. A direct reference is one in which a memory location within the procedure segment has as its value the address of an item in the task segment. If the link map shows modules named $DATA, the routines in the module whose name appears immediately above use direct references. If there is no $DATA, there may be direct references; you need to inspect the listings of the modules to determine if direct references exist.

Indirect references are address-independent. By referencing off a register, the addresses of all volatile data are given to the reentrant procedure segment as a part of the subroutine call. (Many high-level language compilers produce the correct subroutine calls for you.) Reentrant procedure segments can reference constants in the task segment, but must use an indirect reference. Constants embedded within the procedure segment itself can be referenced directly.

Most TI language compilers do not create truly reentrant code (refer to the associated language programmer's guide). With assembly language you can create whatever structures are needed.

2.5.1.2 Address-Dependent Reentrant Procedures. To achieve address-dependent reentrancy, you must correctly link all code and data segments of the program. All code in the procedure must be pure: it must not modify itself. Further, the procedure must have a data segment for the procedure's local data that is directly referenced. The Link Editor can separate the data segment from the procedure segment, and link the data with the task segment.

In assembly language, you must create the segments with the PSEG and DSEG directives, as described in the *Link Editor Reference Manual*. High-level languages automatically produce needed segments for each source module compiled. If you specify the appropriate linking commands, the Link Editor separates the executable code from the data segment in each compiled or assembled module linked. The executable code can be linked into a procedure segment and the data segment into a task segment. This process creates direct references from the procedure to the task since the data items in the data segment are known to the procedure by their memory addresses rather than being passed in the calling sequence. Refer to the *Link Editor Reference Manual* for more information.

While the procedure segment does contain the addresses of data items outside the bounds of the procedure, you can link several tasks in such a way that the address of any data item used by the procedure appears at the same address in all the tasks. Use the ALLOCATE Link Editor command. The procedure will work in reentrant fashion with several tasks, although it is not truly reentrant. Note that replicated copies of the same task from a program file always meet this requirement.

Address-dependent reentrancy is supported by most of the high-level languages available with DX10. Refer to the appropriate language programmer's guide for specific information.

2.5.1.3 Data, or "Dirty" Procedures. If code linked into a procedure contains data within the bounds of the procedure that is modified by task execution, the procedure is referred to as *dirty*. Tasks sharing the procedure cannot assume that the data is unchanged; the task using the procedure last could have modified the data. A task cannot assume that subroutines in the procedure that use the shared data can be called at any time. Some high-level languages available on DX10 do not support any kind of data sharing. Others, like FORTRAN, do support it. Any structure of shared data can be constructed with assembly language.

2.5.2 Sharing Data in Memory

The following paragraphs discuss some specifics of sharing data between tasks.

2.5.2.1 Restrictions on Using Shared Data. Any task executing in the system can be interrupted between any two machine instructions. Therefore, the tasks sharing data must cooperate or synchronize themselves so the data is modified correctly.

Any part of a program accessing shared data susceptible to such corruption is called a critical section. To prevent corruption, a critical section must prevent itself from being interrupted. Under DX10, it is not necessary for tasks to prevent any hardware interrupts.

Several techniques are available under DX10 for preventing interruption of a critical section. Most methods require using available SVCs in a particular way, as described in the following paragraphs. These techniques can usually be implemented in high-level languages also. Refer to the appropriate language programmer's guide for details on implementation from high-level languages.

The simplest technique uses the Do Not Suspend SVC (opcode >09). This SVC prevents any other task from executing until the time interval expires or until the task uses another SVC. Choose a time interval long enough for the critical section to finish its execution without interference from other tasks and short enough to not interfere with the response time of other users. However, with this method the task cannot use any other SVC (including I/O through a language runtime) within its critical section, since using such an SVC automatically terminates the Do Not Suspend status.

If any other SVC is coded within the critical section, you must use another technique. For example, you can use a shared variable to prevent a critical section from being interrupted by another critical section sharing the same data. To employ this technique, each such critical section must employ the following algorithm.

1. Set the shared variable to -1, either with a data initialization statement, or in the first task using the shared data.

2. Before a critical section begins execution, test the variable.

3. If the variable value is -1, set the variable to +1 and go to step 5.

4. If the variable value is already +1, execute the Time Delay SVC (opcode >02) for a short period, and go to step 2.

5. Execute the critical section.

6. Set the variable back to -1.

The test and the set must appear as one operation so that there is no chance of scheduling between the tests and set (steps 2 and 3). Use the Do Not Suspend SVC to prevent interruption between test and set, or use the ABS assembly language instruction. The ABS assembly language instruction tests and sets the variable in one operation. If the variable is negative, it sets the condition code to reflect a negative number before it changes the sign. If the variable is positive, it sets the condition code accordingly but does not change the sign.

NOTE

> You cannot use the ABS function of a higher-level language to perform the test and set, since you cannot access the condition codes and set the variable in one operation. Use an assembly language subroutine or a Do Not Suspend SVC.

This technique keeps a critical section *locked* while the task does any required SVCs. By including an end action routine you can prevent the task from terminating with the lock variable in the locked state (+ 1). Otherwise, all other tasks needing the locked data will not be able to access it and the tasks involved will cease to execute.

If necessary, you can construct queuing mechanisms to handle waiting tasks. These queues should eliminate the need for a task to continually wait and test. Such a scheme usually uses the Do Not Suspend SVC to protect the queues themselves, and uses one of the following pairs of SVCs to handle the tasks:

- Unconditional Wait SVC (opcode >06) and Activate Suspended Task SVC (opcode >07) when a task must wait and when a task finishes using a critical section, respectively.

- Time Delay SVC (opcode >02) and Activate Time Delay Task SVC (opcode >0E) when a task must wait and when a task finishes using a critical section, respectively. This pair has the advantage that the task calling time delay can receive control back after a specified period of time. Such a task can determine when no other task is going to activate it and signal an error condition to the other tasks and/or an operator.

2.5.2.2 Cautions to Observe. There are several cautions to observe when sharing data with tasks that communicate with other tasks.

DX10 has a mechanism for detecting one activate SVC issued for any given task that is not unconditionally suspended. This mechanism is inherent in the Activate Suspended Task SVC (opcode > 07). In other words, this SVC can remember one and only one activate SVC issued, as described in Section 8.

Code the task receiving messages or commands to receive all waiting messages or commands at the time it is activated by a single activate SVC.

To ensure the integrity of the queue, list or other data used, use the Do Not Suspend SVC or the lock variable techniques. The part of the program that manipulates the queue data is a critical section, so the task must issue the Do Not Suspend SVC before it enters that section, and the time delay should be long enough to prevent interruption until the Suspend or Activate Suspended Task SVC completes. The Suspend or Activate Suspended Task SVC should be the last operation of the critical section.

When DX10 performs the Suspend or Activate Suspended Task SVC, the critical section is finished. It is not necessary to keep the queue locked until DX10 returns control to the task after the SVC completes. When queuing messages or maintaining lists in a shared procedure in memory, do not depend on the fact that an Activate Suspended Task SVC will be remembered.

Code the task receiving messages to suspend when its queue is empty. The critical section should include the operations to inspect the queue, determine if it is empty, and execute the Suspend SVC.

2.5.3 Using the Intertask Communication (ITC) Channels

When sending data using ITC channels, it is not necessary to protect the critical section. The operating system provides internal protection of the queue linkage process.

Consider the following example:

A receiving task is interrupted after it has determined the queue is empty, and before it suspends. During the interruption, another task places a message on the queue and executes an Activate Suspended Task SVC. In this instance, DX10 remembers that an Activate Suspended Task SVC was issued while the task was active. Therefore, when the receiving task's Suspend SVC executes, it will not suspend but will be reactivated by the remembered activate SVC.

DX10 ensures the integrity of all other operations with an ITC because the actual queuing and dequeuing operations are performed as SVCs, and cannot be interrupted by the scheduler.

2.5.4 Sharing Procedure Code to Save Memory

Sharing code saves memory and program file space on disk. Saving memory can improve response time because it can decrease the amount of roll-in/roll-out overhead in the system.

To share code, install tasks so that they are attached to the appropriate procedure in the program file. You do this with the Link Editor, as discussed in the *Link Editor Reference Manual*, or use the IT SCI command described in the *DX10 Operations Guide* (Volume II). You can change a task's attachment with the Modify Program Image (MPI) SCI command (also described in Volume II).

When a user task is attached to a procedure, the procedure can be in the same program file as the task, or in the system program file. It cannot be in any other program file.

When you share a procedure, remember that a procedure is known to the system by its ID and the program file from which it was loaded. Only tasks loaded from the same program file can share that procedure, with the following exception: any task can share a procedure on the system program file even if the task itself is installed on another program file. You indicate that a procedure is not on the task's program file when you install the task using the Install Task (IT) SCI command.

2.5.5 User Program Files

User program files are program files that you create yourself for your own applications, using the Create Program File (CFPRO) SCI command. You use them for application-oriented tasks and procedures, rather than using the system program file (S$PROGA). By creating and using your own program files you can do the following:

- Update the release of the operating system more easily. Copy your program file to the new system disk instead of reinstalling all of the tasks and procedures in the new system program file.
- Separate applications to facilitate development. Development can be done on one application without disturbing operational applications.

- Copy selected applications more easily. If you have installed all the tasks and procedures for an application on a separate program file, you only need to copy that file.

Always install tasks in user program files unless the task requires one of the following characteristics:

- Memory resident
- Biddable by the Scheduled Bid Task SVC (opcode >1F)
- Biddable by the Bid Task SVC (opcode >05)
- Nonreplicative, with the installed ID and runtime ID the same

2.5.6 Overlays

Overlays are parts (phases) of a program which share memory with a task. The system loads overlays from a program file into memory during program execution. Overlays reduce the memory needed for a program because the system only loads part of the program initially and then loads overlays as needed during execution.

The memory requirements of a program can be reduced even more if the program uses two or more overlays which share the same overlay area. You can select automatic overlay loading when you link to create code that controls overlay loading. The code is generated in the control, or *root*, phase of the task (which is not overlayed).

Overlays increase execution time by the time required to load them. If you choose frequently-used code for your overlay, the time it takes to load it each time can be a significant part of the total execution time. You can develop an effective scheme by following some simple rules:

- Avoid calling an overlay within a loop that is executed many times in response to a user request.
- Choose overlayed code so that only a very few of the overlays are needed for each function of a program.
- Choose seldom-used code for overlays.

Overlays are only loaded into a task segment, not into procedures.

2.5.6.1 Overlay Structures. Overlay structures are defined by the user and generated by the Link Editor. Refer to the *Link Editor Reference Manual* for information on organizing overlay structures.

2.5.6.2 Overlay Loading. In the root segment or in phases in memory, use the Load Overlay SVC as described in Section 8 to load an overlay from a program file into memory. Since control returns to the instruction after the Load Overlay call, take care that the call itself is not overlayed. The Load Overlay SVC can be issued from a task or a procedure segment, but the SVC call block must be in the task segment.

During link edit you can include an automatic load overlay manager to manage the loading of overlays in a program. Use the LOAD Link Editor command in a link control stream to do this, as described in the *Link Editor Reference Manual*. Some languages require this method. For linking overlays in high-level languages, refer to the appropriate language reference manual for details.

2.5.6.3 Relocatable Overlays. Overlays are usually loaded into memory at the natural load address determined during link edit. However, you may load an overlay elsewhere in memory. Modules installed as relocatable overlays in a program file can be relocated by the load overlay call. This enables users to load overlays where space is available rather than where linked. However, this can be cumbersome, and sometimes difficult to use.

2.5.7 Task Attributes

Tasks can be installed with one or more of the following attributes specified:

- Privileged/nonprivileged
- System task/user task
- Priority
- Disk resident/memory resident
- Replicatable/nonreplicatable
- Arithmetic overflow protection (990/12 only)
- Execute protection (990/12 only)

You can install procedures with some of these same attributes. Each of these attributes and their advantages and effects are discussed in the following paragraphs.

2.5.7.1 Privileged and Nonprivileged Tasks. Most user tasks are nonprivileged and are therefore prohibited from executing certain system functions. A task is specified as privileged or nonprivileged by a parameter in the IT SCI command, or by a parameter in the Install Task SVC.

You must install a task as privileged when that task requires use of privileged SVCs, or if you want to execute privileged machine instructions. Otherwise, install it as nonprivileged.

CAUTION

Tasks installed as privileged have the potential of destroying the operating system. Tasks should not be privileged unless absolutely necessary.

2.5.7.2 System and User Tasks. A task is either a system task or a user task. System tasks have the following characteristics:

- Execute in privileged mode
- Execute in system memory address space (coexistent with other portions of the system)
- Maximum of 16,384 bytes in logical length

Cautions given for privileged tasks also apply to system tasks. In fact, they are more dangerous to the operating system since they execute in system memory space. There should be no need for application-oriented tasks to be system tasks.

2.5.7.3 Priority. When a task is installed, you must assign a priority level that is used by DX10 to schedule the task each time it is activated, or when rescheduling occurs. (Priority and scheduling were discussed in detail earlier in this section.)

2.5.7.4 Disk Resident and Memory Resident Tasks. Tasks may be memory-resident or disk-resident. Memory-resident tasks are always in memory, whether executing, suspended, or terminated. Disk-resident tasks are in memory when executing, and can be in memory while waiting for execution. However, they can also be rolled to disk when not executing to provide memory space for another task that needs to be rolled in from disk for execution. Most user tasks should be disk-resident to free memory for other tasks. Also, some SVCs depend on the DX10 roll-in/roll-out facility.

Memory-resident tasks must be installed on the system program file and are effectively disk-resident tasks until the system is rebooted. Certain support features which depend on roll-in/roll-out (such as dynamic memory allocation) are not available to a memory-resident task.

Once resident, memory-resident tasks occupy memory, even if terminated, until they are deleted from the system program file or made non-resident, and the system is rebooted. Residency is a parameter supplied when the task is installed on the program file.

2.5.7.5 Replicatable and Nonreplicatable Tasks. Tasks specified to be replicatable can have multiple copies concurrently in memory. Replicatable tasks are frequently used in multiterminal systems, allocating one (or more) copies for each terminal. Replicatability is a parameter that is supplied at task installation.

If a task is replicatable and memory resident, the first request to bid the task activates the memory-resident copy. If that copy is not in state 4 and the task is replicatable, additional invocations function exactly like replicated disk-resident tasks.

2.5.7.6 Arithmetic Overflow Protection. The 990/12 allows detection of arithmetic operations that cause an overflow or underflow. This error condition is signaled as a task error.

2.5.7.7 Execute Protection. The 990/12 allows execute protection of task segments and execute and/or write protection of procedure segments. Executing an execute protected task segment or executing or writing to a protected procedure segment causes a task error.

2.5.8 Programming Prohibitions

Tasks installed in DX10 are either privileged or nonprivileged. Privilege is enforced by DX10 and the computer hardware. Most programs are nonprivileged and may not use the following assembly language instructions:

- Computer control instructions (RSET, IDLE, LREX, LIMI)
- Real time clock instructions (CKON, CKOF)
- Mapping instructions (LMF, LDD, LDS)
- I/O instructions (SBO, SBZ, TB, LDCR, STCR) when the CRU address is greater than >0E00

Further, nonprivileged tasks may not use the following privileged supervisor calls (SVCs):

Install Disk Volume	Delete Task
Unload Disk Volume	Delete Procedure
Initialize Disk Volume	Delete Overlay
Allocate Disk Space	Assign Space on Program File
Direct Disk I/O	Get System Pointer Table Address
Open File Unblocked	Suspend Awaiting Queue Input
Install Task	Kill Task
Install Procedure	Read/Write Task
Install Overlay	Read/Write TSB
Initialize Date and Time	Abort I/O

Finally, no nonprivileged task may access memory outside its assigned memory space. Privileged tasks can do so, but extreme care must be taken to ensure no damage occurs to the operating system. Nonprivileged tasks can request additional memory using the Get Memory supervisor call, provided the total program size remains less than or equal to 65,504 bytes, and the task is not memory-resident.

Any prohibited access to instructions, supervisor calls, or memory causes a fatal task error. DX10 aborts the task and transfers execution to the task's end-action entry point if end action is specified.

NOTE

Volume V contains all privileged SVC code descriptions.

2.6 TASK TERMINATION

Tasks executing under DX10 may terminate normally or abnormally. In either case, the task should make provisions for termination. If a task does not explicitly invoke termination, it either loops infinitely or it attempts to violate its memory bounds, causing abnormal termination.

2.6.1 Normal Termination

To terminate normally, a task executes an End Task SVC. (All high-level languages have a stop or end statement to cause an End Task SVC to be executed.) DX10 then releases the task's resources and takes it out of execution. Disk-resident tasks disappear from memory. Memory-resident tasks remain in memory and occupy space but do not execute.

2.6.2 Abnormal Termination

If a task commits a fatal error (illegal instruction, supervisor call, or memory reference), two things can happen:

- If the task transfer vector includes an end action address, the routine at that address is activated. Typically, the end action routine executes an End Action Status supervisor call, returns the data, and executes an End Task supervisor call. It is possible for the end action routine to implement error recovery procedures. When an end action routine commits errors, DX10 unconditionally aborts the task unless a Reset End Action SVC is issued.

- If the task transfer vector entry for end action is less than 16, DX10 unconditionally aborts the task.

You can kill a foreground task externally by issuing the Kill Task (KT) SCI command from another terminal, or by pressing first the Attention key, releasing it, and then holding down the Control key while you press the X key at the affected terminal. To kill a background task, issue the KT command from any terminal, or the Kill Background Task (KBT) command from the terminal that originated the background task.

In either case, the error is reported to the system log.

In higher-level language programs, the transfer vector and end action address is set up either by the runtime or by the compiler in the main program. High-level language programmers do not need to explicitly code an end action address, and usually do not have any control over them. Refer to the applicable language manual.

NOTE

Throughout this manual, the names of keys are generic key names. In some cases, the names on the keycaps of the terminals match the generic key names, but in many cases they do not. Appendix A contains a table of key equivalents to identify the specific keys on the terminal you are using. Drawings that show the layout of the keyboard of each type of terminal are also included.

3

DX10 I/O System

3.1 INTRODUCTION

The DX10 I/O support system facilitates disk file I/O for application and system programs. This support includes managing file I/O requests and disk file space, and controlling device allocations. Disk file I/O is performed through supervisor calls (SVCs) to the operating system, either directly, or through high-level language runtime facilities.

SVCs are requests for operating system services. Assembly language programmers must code SVC call blocks directly by coding a call block to include the hexadecimal SVC code, and the additional parameters required to perform the service. Many of these same services are supported by high-level languages. They are still performed by calls to the operating system for file services, but the calls are generated by the language runtime routines. The calls are transparent to the user. An example of an SVC in a high level language is a READ statement. In COBOL, the READ statement requests specific services from the operating system, namely, retrieving information from a file. SVCs are discussed in detail in Sections 7, 8, 9, and 10.

The remainder of this section discusses the supported file types and their usage. The internal system utilities supporting the DX10 file system, disk file logical and physical organization, and device and LUNO usage and management are also described.

3.2 SUPPORTED FILE TYPES AND USAGE

DX10 supports three different file types:

- Sequential files
- Relative record files
- Key indexed files

The following paragraphs discuss the basic organization of each file type and when each file type would be most advantageous to use in an application. Information on how DX10 allocates space for files, how to choose logical and physical record length, and how DX10 manages files and disk space are discussed in later subsections.

3.2.1 Sequential Files

Records in sequential files must be accessed in the order that they appear in the file. They can only be placed in the file in sequential order.

Even though there is no random access on sequential files, keys can be imbedded in the record and used to identify a particular record. Therefore, "random" access can be simulated by reading through the file and testing each key to determine if it is the key you desire. Every record must be processed during the search, making this random access on sequential files very inefficient in most circumstances. If other advantages of sequential files are required, such as maximum space usage, occasional "random" searches are acceptable.

Sequential files are advantageous in applications where rapid input or output of all records in the file is required. Data being sent to the line printer or terminal with each record representing a line of the report or text is an example. A sequential file is also a good format for storing data, since it takes comparatively less space than other file formats, and can be stored on sequential media such as magnetic tape. This requires less space because there is no unused disk space as is possible in relative record formats. Also, there is no space dedicated to a key table as required for key indexed files. Blank suppression and adjustment can be applied to sequential files to minimize the space required for each record. Also, record-locking is supported. (These attributes are described in later subsections.)

Examples of sequential file use include the following:

- Log files to which records are written in sequential order
- Output files generated by applications programs such as written reports
- Any output intended for the line printer
- Source programs
- Listing files from such DX10 processes as the assembler and Link Editor
- Input files of card images (if a logical record length of 80 is specified for the file, the file then "looks" just like data from a card reader to the program reading the file)

Sequential files are always blocked for efficient file and disk management. Blocking is discussed later in this section.

3.2.2 Relative Record Files

Relative record files can be read sequentially, or accessed randomly. Each record in a relative record file is addressed by a unique record number. The record number represents the relative position in the file. For example, record number 10 is located in the tenth record position in the file.

Unlike sequential and key indexed files, the space for a record is reserved in the file whether or not a record with that number actually exists. Your record numbers should be tightly packed if you use relative record files to conserve file space. Relative record numbers range from 0 to one less than the number of records in the file. (Some high-level languages adopt a different convention, handled by the runtime package. Refer to the individual language manuals.)

The upper limit on the number of records in a relative record file is 2^{24}. You can make your data compatible with the record numbering by writing your program to compensate for the difference. For example, if you have inventory part numbers of closely packed or consecutive numerical values, but they are ten digits long, include a calculation in your program to subtract the beginning part number in the series from whatever part number is specified for access.

DX10 converts the record number to a physical address on the disk (track and sector) and can directly access any record in one disk access. Conversely, relative record files may be accessed sequentially by specifying a starting value in the record number field. DX10 automatically increments the record number after each read or write.

Relative record files can be blocked or unblocked, as discussed later. Program files, image files and directory files are unblocked relative record files used as special-purpose files. They are accessed differently than blocked files. Unblocked files do not depend on blocking buffers external to the program to manage read/write operations.

3.2.3 Key Indexed Files

A key indexed file allows random access to its records through a primary key value, with up to 13 secondary key values. A key is a character string in a fixed position within the record. For example, records in an employee file can be accessible by employee ID, employee name, and employee social security number. Each record could begin with the employee ID, then the name (in a fixed number of characters) and then the social security number. These fields would be set up as key fields.

In addition to random access, key indexed files have these features:

- Records may be accessed sequentially in the order of the key values of any key field. Functionally, this gives the same access as if they had been sorted in that order.
- Key values can have duplicates; that is, two or more records in the file can have the same value for a primary or secondary key. (You specify whether to allow duplicate keys when you create the file.)
- You can modify secondary key values. This means that you can read a record, change the key value and rewrite the record. You can also add key values previously missing from a record. The social security number of an employee can be added, for example. (You cannot modify and then rewrite primary key values, however. Primary keys are always unmodifiable.)
- Key values for secondary keys can be null, causing the record to not be catalogued in the index for that key. A key value is null when it is all blanks or has a value of >FF in the first byte. A key value containing all binary zeroes is not null.
- Keys may overlap. For example, you can use the social security number as a secondary key, while using a job code and the first three digits of the social security number as an employee ID.
- A key may be up to 100 contiguous characters in length.
- Records may be of variable length and may change in size on a rewrite. (However, records may not have a length of zero or an odd number of characters.)
- Positioning on partial keys is allowed.
- Records are automatically blank suppressed. (Blank suppression is discussed later.)

- Record level locking is supported. (Record locking is discussed later.)
- Integrity of the files is maintained through preimage logging of modified blocks. Before the system modifies a physical record, it copies the record to a backup area in the file overhead area. System crashes and power failures can only result in loss of the last I/O operation, unless the logging characteristics have been modified using the MKL command.

Key indexed files are always blocked for maximum disk access efficiency. Generally, they are the most advantageous files for general-purpose applications data. Some tradeoffs are involved, as discussed in the following paragraphs.

3.2.4 File Usage Tradeoffs

Each file type (sequential, relative record, and KIF) offers certain advantages for different applications. Table 3-1 shows some of the tradeoffs associated with each file type. The following paragraphs discuss these tradeoffs. This information represents general guidelines only. For detailed information about file use guidelines for a particular language, refer to the programmer's guide for that language.

Table 3-1. File Usage Tradeoffs

Type	Disk I/Os Required	Format	Utilization
KIF	Average of 3-5 I/Os per logical operation to retrieve record.	Maximum 14 keys (alphanumeric). Each can be up to 100 bytes.	Must allocate disk space for key tables. For widely dispersed (random) key values.
Relative	At most, one per logical operation.	Record number is internally formatted as a three-byte binary integer.	No key table, but allocates space for unused records. Good for tightly packed numeric keys.
Sequential	At most, one per logical operation.	No key.	Space efficient for variable-length records. Good when records are to be accessed in order.

3.2.4.1 Sequential Files Versus Relative Record Files. Sequential files use disk space more efficiently than relative record files when variable-length records are accessed in order. Relative record files always store records as fixed length. When random access is required, relative record files are more efficient, since sequential files must be forward spaced or back spaced to access specific records. The number of physical disk events on forward/back spacing operations can be reduced by blocking (discussed later). If disk space is not a consideration, relative record files are more efficient for applications requiring more than occasional random access. Blank suppression and adjustment in sequential files can also save space.

3.2.4.2 Relative Record Files Versus KIFs. Although KIFs are more efficient in most applications than relative record files, KIFs require additional physical disk events per logical I/O operation. Three major factors determine which of these two file types is best. These factors are as follows:

- Key format
  - If you require an alphanumeric key, you must use KIF. Relative record file keys must fit into three binary bytes and be numeric only.
  - You can search a KIF for a partial key if you know part of the key value, and search on any of up to 14 keys. However, relative record files are randomly accessed by record number. If the record number is not known, search sequentially, or code the program to use another search pattern.
- File space utilization
  - Relative record files allocate space for records from zero to the largest key in the file regardless of the number of records actually used. If the record density in the file is normally above 70 percent, this is an efficient file type, considering access speed (discussed below). Compute the density percentage by dividing the number of records used by the largest record number on the file, then multiply by 100.
  - KIFs require substantial overhead space for key tables but allocate file space only for records that are in the file. If the file has widely dispersed key values, KIF should be selected.
- Access speed
  - Relative record files require only one physical disk I/O event for each read or write operation (zero if the file is blocked and the logical record is already in memory).
  - KIFs with single keys usually require less than six disk I/O events per logical operation when the mix of logical operations includes more reads and rewrites than writes.
  - KIFs usually require from three to five disk I/O events for each random read operation, and can require over 100 disk I/O events for add operations on files with multiple keys.

In general, if speed is the overriding consideration, then use relative record files (assuming the key format is practical). If space is a consideration, the file record density must be taken into account. If speed or file space are not considerations, select the most convenient file type.

3.3 FILE FEATURES

Certain features are available on DX10 that enhance file efficiency or file protection. The following paragraphs outline these features.

3.3.1 Blank Suppression and Adjustment

Records in a file can be compressed into a smaller space by suppressing the blank characters in the record. Blank suppression and blank adjustment save disk space within a file by storing data in a more compact form. Since this essentially makes the record length variable (depending on the number of blanks in each record), these techniques apply only to sequential files and KIFs. They cannot be applied to relative record files.

Blank suppression replaces strings of blanks by a count of blanks when writing to disk and restores the blank string when reading from disk. The blank suppression operation is done by the system, and is transparent to you. It is generally advantageous to specify blank suppression for all files that usually contain many blanks, such as:

- Source files
- Listing files
- Text files

It is less advantageous to use blank suppression for files that generally have very few blanks. A blank-suppressed record with no blanks is two bytes longer. Examples of such files are:

- Binary files (not ASCII data)
- Relocatable ASCII coded object files

KIFs are automatically blank suppressed. You can specify suppression for sequential files when you create them.

The second method of blank compression is called blank adjustment. Blank adjustment truncates trailing blanks on output and restores them on input. Besides KIFs and sequential files, blank adjustment can be applied to I/O devices with variable record lengths.

You can implement blank adjustment in KIFs or sequential files by setting the bit in the I/O call block. (In high-level languages that support this feature, blank adjustment is controlled by the run-time package.)

3.3.2 Expandable Files

When you create a file (using either the SCI commands or the Create I/O SVC), you specify the initial or primary allocation parameter that indicates the initial file size. You can also specify a secondary allocation parameter for use by the system to calculate the file space needed if the file outgrows the initial file space. For sequential and relative record files, specify the file to be *expandable* when you create it if you want the system to automatically allocate secondary disk space. (KIFs are always expandable; there is no expandability parameter in the create file command for KIFs.)

When the file grows to exceed its primary allocation, it is augmented with secondary allocations. If you designate the file as expandable, and do not provide a secondary allocation parameter, the system takes a default parameter. Later paragraphs in this section describe secondary allocation algorithms.

If you do not want a sequential or relative record file to expand beyond the initial allocation, specify NO to the EXPANDABLE? prompt when you create the file.

3.3.3 End-of-File (EOF)

An EOF mark for a file is a logical position within the file that indicates the end of the file. When a read operation encounters an EOF mark, the EOF status bit is set (bit two of the system flags). No data is transferred. Key indexed files have no EOF mark. Relative record files have exactly one EOF mark, which corresponds to the record following the highest-numbered written record.

Sequential files can have multiple EOF marks. Thus, a sequential file may consist of multiple data sets or subfiles delimited by EOFs allowing it to function similarly to a magnetic tape containing several different files. As with magnetic tapes, you must space forward until you encounter an EOF mark, or the required number of EOF marks, to indicate when you have encountered the desired subfile before processing. An attempt to read beyond the last EOF in a file results in an error code of >30.

EOFs are internally represented in sequential files by records uniquely recognized as an EOF mark. A Write EOF or Close with EOF SVC writes the EOF mark on the file. Writing an EOF does not prevent writing more records into the file.

3.3.4 File and Record Protection Features

DX10 provides several features for protecting files from program flaws which might otherwise destroy valuable data. These protection features are:

- Delete and write protection
- Record locking
- File access privileges
- Immediate write attribute (forced write)

The system protects special usage files (program files, image files, and directory files) from being accidentally destroyed by preventing a LUNO from being assigned to them without verification.

Each file protection feature is described in the following paragraphs. Except for delete and write protection, these features are usually handled by the runtime package for high-level languages. For information on how to implement these features in assembly language, refer to Section 5.

3.3.4.1 Delete and Write Protection. Two of the file protection features provided by DX10 are delete and write protection. These file attributes are modifiable by standard I/O SVC calls. Files are initially created without protection. A subsequent SVC call must be made to invoke protection. High-level language runtimes do not support delete and write protection. You can set these yourself using the Modify File Protection (MFP) SCI command.

An attempt to write to or delete a file with write protection will fail and return an error code. An attempt to delete a file with delete protection will fail and return an error code. Write protection includes automatic delete protection. These protective attributes are not intended for file security. However, they provide protection against program flaws and operator errors that might otherwise destroy valuable data. You can remove write and delete protection using the appropriate non-privileged SVC.

NOTE

Write protection and delete protection applied to directories do not protect the files within that directory, but only protect the directory node itself.

3.3.4.2 Record Locking. DX10 provides record locking for files. Basically, access to a given file may be shared between several users, yet individual records may be locked to provide exclusive (single user) read and write access. This is not a security feature since any file user can unlock a locked record. However, this feature is necessary to ensure that record updates occur one at a time. For example, inventory files might be accessible from several terminals. Record locking can prevent two or more users from updating a record simultaneously, causing an undetected loss of one of the updates.

The following examples illustrate this feature. Without record locking, the following update activity can occur:

1. User A reads a record.

2. User B reads the same record.

3. User A updates the copy of the record and writes the updated record to disk.

4. User B updates this copy of the record and writes these updates to disk, destroying User A's updates to that record.

With record locking, this same activity occurs as follows:

1. User A reads a record. The Lock/Unlock flag in the call block (byte 5, bit 5) is set to lock the record.

2. User B attempts to read the same record but, finding the record locked, waits for user A to unlock the record before proceeding.

3. User A updates the record, writes it back to disk which unlocks it.

4. User B reads the record and locks it.

5. User B updates the record, writes it back to disk and unlocks it. Both updates are now included in the record.

In assembly language programs, implement record locking using an I/O SVC. In high-level languages that have suitable syntax, use the language to implement record locking. If the language does not support record locking, you must code an SVC to do so. Refer to the applicable language manual.

NOTE

> When program A has a record locked and program B attempts to read the record, the record pointer for sequential and relative record files points to the record that is locked. For a key indexed file, the currency is updated so that it is possible to read past the locked record.

3.3.4.3 File Access Privileges. DX10 supports several different access modes, or access privileges. These access privileges define the relationship between logical units and files and disallow conflicting accesses by other logical units.

3.3.4.3 File Access Privileges. DX10 supports several different access modes, or access privileges. These access privileges define the relationship between logical units and files and disallow conflicting accesses by other logical units.

There are four access modes applicable to files, as follows:

- Read Only — Allows more than one program to read from the file, but a program with this access mode cannot write to the file.

- Share — Allows more than one program to read, write, or rewrite to the file. (When the file is sequential, Share allows both read and rewrite operations, but not write operations.)

- Exclusive Write — Allows more than one program to read the file, but only the program with this access mode can write to the file.

- Exclusive All — Allows the program with this access mode to read, write, or rewrite to the file, and no other program can access the file.

In assembly language programs, you specify the access privilege when you code the SVC call block for the Open, Open Rewind, or Open Random I/O SVC operations. Set bits three and four in the user flags byte of the call block as follows:

Code	Privilege
00	Exclusive Write
01	Exclusive All
10	Share
11	Read Only

For high-level languages, refer to the appropriate language programmer's guide and language reference manual to determine how these access privileges are implemented.

Two programs accessing the same file must be coded with compatible access privileges. If a program executes one of these SVCs, and an inconsistency results, the SVC generates an error indicating a conflict in access privileges. Also, you can change access privileges during program execution using an I/O SVC, as long as the change does not cause an inconsistency.

Table 3-2 illustrates the allowed and forbidden combinations of access privileges. In the table, "A" indicates an allowed combination, and "I" indicates an inconsistent combination.

Table 3-2. File Access Mode Compatibility

Relative Record and Key Indexed Files

	Read Only	**Shared***	**Exclusive Write**	**Exclusive All**
Read Only	A	A	A	I
Shared	A	A	I	I
Exclusive Write	A	I	I	I
Exclusive All	I	I	I	I

A — Allowed combination.
I — Inconsistent combination.

Note:

* Shared sequential files allow read and rewrite only.

3.3.4.4 Immediate or Forced Write. When DX10 reads records from a blocked file into memory, the file block containing the requested record remains in memory as long as possible. Subsequent read and write requests read and write from and to the memory-buffered file block, and not the disk. Only when DX10 needs the memory area will the disk be accessed. Since memory is far faster than disk, deferring of disk writes increases system throughput. However, since the disk write operation is reported complete and yet is actually deferred, any errors which occur during the write cycle will be unexpected and in some situations may be undetected by the user. For this reason, a forced write option is provided at file creation. This file attribute prevents disk write operations from being deferred.

The most common undetected error is disk failure. A user could update a record in a block and be informed that the update has been successfully completed. However, when the block is actually written to disk, possibly several minutes later, an I/O error could occur. This error is returned on the next supervisor call made to the LUNO after the error. The error is returned even if the service call is not a write operation.

Undetected errors are rare and files with the immediate write attribute are less efficiently processed, especially sequential files. Therefore, the user should reserve this attribute for sensitive files where the loss of small amounts of data is not permissible. Key indexed files always have the immediate write attribute, subject to options selectable by the MKL SCI command.

To implement the Immediate Write attribute, specify it as an attribute when you create the file (either using a CF or the Create File SVC).

3.3.4.5 Special Usage File Protection. You can protect special usage files such as program files, directory files and image files, from accidental use by setting two flags in the SVC call block to zero when assigning LUNOs to files. (You can create a call block in high-level languages to do this, but runtimes do not set these two flags.) The flags indicate whether the LUNO is being assigned to one of the special-usage files, or to a sequential, KIF or standard relative record file. (Special usage files are nonstandard relative record files.) Then, when a program requests a file assignment, the file management utilities check the flags to determine the type of file to be accessed. If these bits are zero and the file is a special usage file, the LUNO is not assigned and an error results. Otherwise, the SVC operation completes the Assign LUNO operation. The flags are bits 1 and 2 of byte 16 in the call block of the Assign LUNO SVC. (Refer to Section 10, SVC subopcode >91, for coding information.)

3.3.5 How the System Handles File I/O

DX10 uses a queuing mechanism to service file requests. It is possible to have more than one file operation going concurrently to different files (by creating more than one file manager task at system generation time), but DX10 strictly controls operations to each individual file with a first in first out queue. Thus, only one operation is performed on any given file at any given time. All requests for service on a file that currently has a request active will be queued and processed one at a time as each previous request completes.

3.4 DISK FILE ORGANIZATION AND MANAGEMENT

Generally speaking, all DX10 files are disk files. They reside on disk, and portions of them are read into memory as they are needed by various system and user tasks.

All files have certain organizational characteristics in common. Also, each file type has a few unique organizational characteristics. These organizational characteristics determine how DX10 manages I/O requests and disk space for the file.

The DX10 operating system uses three subsystems to handle the required functions of the file system. These subsystems, governed by the operating system, and transparent to the user, are the file utility, the file manager, and the disk manager.

The file utility (FUTIL) is the part of the system that performs assign LUNO calls (for devices as well as files), create file calls, and other functions, in response to calls from user programs. It performs them one at a time in a first in first out order. If the I/O SVC subopcode is greater than or equal to >90 and less than or equal to >9F, the file utility subsystem processes the call.

The disk manager (DSKMGR) is the part of the system that performs disk allocation. It finds disk space in ADUs in response to requests from FUTIL and FILMGR, and determines how physical records will be blocked into ADUs. This part of the system sets bits in the bit maps on allocation and clears them on deallocation. Deallocation only occurs on file deletes. (An internal call mechanism is used.)

The file manager (FILMGR) is the part of the system that performs all file I/O in response to requests from user programs and other system tasks. I/O SVC subopcodes in the range 0 through >14, and >4A for files that are not KIFs, are handled by the main part of FILMGR. All subopcodes for KIFs, including the sequential subopcodes supported, are handled by the key indexed file manager, which is an optional part of the file manager selected during system generation. This part (FILMGR and KIF, if used) handles blocking of logical records into physical records and determines how much disk space to request when a file expands.

The remainder of this section describes the basics of DX10 file organization, file management techniques implemented by the file utilities, and how organization and management are related to efficient disk file usage.

3.4.1 File Management Strategy

The DX10 file management strategy is designed to meet three performance objectives:

- To provide access to any physical record of the file using one disk access
- To provide for wide dynamic range of file size without incurring excessive allocation overhead
- To provide for efficient use of disk storage space

The first objective relates to logical records as well as physical records. Typically, there are several logical records stored within a physical record, as explained later. If the file management system can access any physical record in a single disk access operation, logical records within the file can often be accessed in a single disk operation, making file I/O for programs more efficient. Exceptions include key indexed files where several disk accesses are sometimes required to read or write a logical record to or from the disk file. (Logical and physical records are discussed in detail later in this section.)

The second objective relates to allowing files to grow in size, and still maintain an efficient way of tracking their physical location on disk. As files grow and require more space, sometimes the space needed exceeds the size of any available segment. The file gets segmented, occupying several smaller, noncontiguous segments of the available space, rather than one or two larger segments. The DX10 disk management provides a memory-resident directory list called the File Control Block (FCB) that catalogues all segments of disk space allocated to the file. The size of the allocated segments varies, but always corresponds to ADU boundaries.

Allocation overhead refers to the following:

- The time spent in the allocation function
- The disk space wasted when the file does not use all the space allocated (this is possible since allocations are always in ADUs)
- The memory space used to catalog allocated disk segments

A successful disk management strategy depends on how well the disk access methods mesh with the physical attributes of the physical disk to provide efficiency in disk storage. The following paragraphs discuss the relationship between the disk management strategy and disk access methods in detail.

3.4.2 Physical Disk Structure

To understand DX10's disk management system, you must understand the physical disk structure and how DX10 manages ADUs, since DX10's disk management methods depend on these structures. DX10 supports several types of physical disk media. All disk media have certain attributes in common. For example, all are physically divided into sectors, and space on each of them is allocated in units of a specific size. The size of these sectors and the number of sectors in one allocatable unit differ among the disk types. The following paragraphs define the physical attributes of disks in general, and provide the details on the physical attributes of each type of disk supported by DX10.

3.4.2.1 Disk Sectors. Disks are physically formatted into sectors. When you initialize a new disk using the Initialize Disk Surface (IDS) SCI command, you are formatting the disk into the proper format required for DX10. Sectors are like records to a disk. All tracks are initialized into a one-sector-per-record size. A one-sector-per-record format defines a sector of data as the minimum amount of data that the hardware can transfer on any given I/O operation. This record is a characteristic of the type of disk and is not necessarily the physical record size for files created on the disk. However, the disk hardware may transfer multiple hardware records (sectors) of data. It is desirable for the physical records and ADUs referenced by the operating system and application programs to be integral multiples of the hardware records (sectors) in size. This is discussed in more detail later.

3.4.2.2 Allocatable Disk Units (ADUs). An ADU is the unit of space that is the smallest unit that can be allocated by the system to a disk file (for primary or secondary allocation). ADU size is dependent on the type of disk, and is comprised of one or more whole sectors, depending on the disk type. ADUs have the following characteristics:

- An ADU always starts on a sector boundary.
- There are no sectors wasted between ADUs. However, if the total number of sectors on a disk is not a multiple of the number of sectors in an ADU, there will be a few sectors wasted at the very end of the disk.
- An ADU may extend over two adjacent tracks.
- ADU size is chosen so that several criteria are met:
  - It is either one sector in size or a multiple of three sectors.
  - The total number of ADUs on a disk is less than or equal to 65535.
  - Partial bit maps for allocating all ADUs will fit on track 0 in the space available for bit maps. Two sectors are always used for volume and bad track information.

3.4.2.3 Format Information for Supported Disks. Table 3-3 shows ADU, sector, and other format information for disks supported by DX10.

Table 3-3. Format Information for Supported Disks

Disk Type	Heads per Drive	Units per Drive	Tracks per Disk	Sectors per Track	Sectors per ADU	Bytes per ADU
FD1000	2	1	154	26	1	288
DS31	2	1	406	24	1	288
DS10	4	2	1,632	20	1	288
DS25	5	1	2,040	38	3	864
DS50	5	1	4,075	38	3	864
DS80	5	1	4,015	61	6	1536
DS200	19	1	15,485	38	9	2592
DS300	19	1	15,257	61	15	3840
CD1400-32	2	2	821	64	1	256
CD1400-96 (rem)	1	1	821	64	1	256
CD1400-96 (fix)	5	1	4,105	64	6	1536
WD500-5	4	1	600	32	1	256
WD500-10	4	2	600	32	1	256
WD500A	3	1	2,082	32	3	256
WD800-18	3	1	1,953	37	3	768
WD800-43	7	1	4,557	37	3	768
WD800A/38	5	1	4,555	33	3	768
WD800A/69	9	1	8,199	33	6	1536
WD800A/114	15	1	13,560	33	9	2304
WD900-138	10	1	8,050	67	9	2304
WD900-138/2	10	2	4,025	67	6	1536
WD900-425	24	1	16,636	100	27	6912
WD900-425/2	24	2	8,316	100	15	3840

On a new disk, all of the space except for tracks 0 and 1 are available for file space. The used or *overhead* portions of the disk contain such information as volume name, location of the VCATALOG, bad ADU list space, and bit maps containing ADU allocation information. The ADU bit maps contain the allocation flags for each ADU, which indicate whether that ADU is allocated to a file (*occupied*), or non-allocated (*free*).

3.4.3 File Structure

Files on disk are composed of physical records. Physical records are composed of logical records that correspond to the data records you store in a disk file and access with application programs. Logical record usage is closely related to the application requiring the data stored in the file. Physical records are related to the physical structure of the disk. Efficient disk access is related to the ratio of logical records to physical records that you specify when you create a file. Efficient disk storage is related to the ratio of physical record size to ADU size (an integral multiplier or division), as discussed in the paragraphs on choosing physical and logical record sizes.

3.4.4 Logical Records

A file consists of a collection of data groupings called logical records. They are called *logical* records because each one represents the unit of information that can be read or written by a user program with one SVC. This division of the file into logical records does not necessarily correspond to the physical division of data on the disk, as discussed later.

The length of the logical records within a file can be constant, or it may vary from record to record in the file. These two file length possibilities are often referred to as *fixed length* and *variable length*. File types differ in the record lengths they allow, as follows:

- Sequential files — Allow both fixed and variable length records
- Key indexed files — Allow both fixed and variable length records
- Relative record files — Allow only fixed length records

You do not have to specify whether the records will be fixed or variable when you create the file. However, you must furnish a logical record length parameter for use by the system to allocate disk space. The following paragraphs discuss how the system uses the logical record length parameter in managing file space, and the constraints on selecting a logical record length for each file type.

3.4.4.1 Constraints for Sequential Files. The system uses the logical record length you specify when you create a sequential file for the following purposes:

- Computation of initial and secondary allocations for the file
- Returning the logical record length to programs via Read Characteristics and Open SVC calls

Since logical records in sequential files can be of variable length, the record length you specify should be an average length of the records that will occupy the file. Using this average length, the system can allocate file space.

Although the system makes no checks on the length of information you place in a logical record, you must create a sequential file with a logical record length less than or equal to the physical record length. Since sequential files are always blocked, there is no gain in making the logical and physical record lengths the same. In fact, there can be considerable waste. In general, a good logical record length for sequential files is one that permits at least three logical records per physical record. However, the best rule of thumb is: choose the logical record length according to the requirements of the application.

System handling of sequential files is as follows:

- Records written to the file may be any length, including zero or an odd number of bytes. They can also be longer than the logical record length specified when the file was created.
- The system makes no check that the written record length matches the record length the file was created with. Any regulation of record length must be done by the application program, or possibly by the high-level language runtime.
- Almost any length can be written, limited only by the address space the file management system needs to block and unblock the records.
- Whatever length is written, if a sufficient number of characters is requested on a read, the number available will be returned. There is no error or other indication if the record contained more characters than requested. Any extra characters are passed over by the system.

- A logical record occupies as many physical records as needed. Usually, several logical records occupy one physical record.
- Logical records can span physical record boundaries. Therefore, there is no wasted space inside physical records.
- Deleted records are not supported by the system. Any application that must use deleted records must do so by writing a flag value in the record. (Existing records can be deleted by using the rewrite option and changing a character in the record that your program recognizes as meaning do not process the record.)
- The system supports rewrite operations of records in sequential files. However, the new record must be of the same length as the old record. In a binary file (not blank suppressed), this is relatively easy to control. In a blank suppressed file, the length after blank suppression must be the same, which means it is much more difficult to control. Any change affecting the position of blank fields can result in an error, even if the total length of blank fields and the total length of nonblank fields is held constant.
- When a sequential file is created, only the first disk sector of the file is cleared to zero.
- Any write operation establishes a new end-of-medium and renders any records past the point of the write inaccessible.

3.4.4.2 Constraints for Relative Record Files. The system uses the logical record length you specify when you create a relative record file for the following purposes:

- Computation of the position of the record within the file
- Computation of the initial and secondary allocations for the file
- Returning the logical record length to programs via Read Characteristics and Open calls
- Limiting the length of data written to the file (since the records are fixed length records)
- Padding the length of data written (if less than the designated fixed length)

Relative record files must have fixed length records because of the organization of the file. A record occupies the position in the file based on its record number. Space is allocated for all record numbers from 0 to the highest number in the file. Fixed length records make it possible for the system to calculate the physical position of any logical record written on the disk, relative to the beginning of the file. Essentially, the location on the disk is determined by the system using the following calculation for relative record files:

Logical record position =

File position + (record number × record length)

In other words, the system multiplies the fixed logical record length by the record number to determine how many bytes from the beginning of the file the record is located. (The beginning of the file is always known by the operating system.)

When you create the file, specify a logical record length that is less than or equal to the physical record length. If exactly equal, the file is unblocked; otherwise it is blocked, with the first logical record of the file beginning at the beginning of the first physical record.

System handling of relative record files is as follows:

- Records written to the file must be an even numerical length.
- Records written to the file are always the same length (fixed length). If the length specified by the write operation is too long, the data is truncated. If too short, it is filled on the right with null characters (binary zeros). There is one exception to this rule: if an unblocked relative record file has records long enough for two sectors to be occupied by one record, and a record is written with enough characters to only fill part of the first sector, the second sector is unmodified.
- Records read cannot exceed the created logical record length.
- In a blocked relative record file, logical records do not span physical record boundaries. Hence, space can be wasted inside physical records if the physical record size is not an exact multiple of the logical record size.
- Unblocked relative record files do not use the system's file buffering and blocking mechanism (described in the subsection on blocking). Therefore, they are not subject to the limitations imposed by the address space of the file system. Unblocked records may be up to 32,766 bytes in length.
- The system does not clear any of the disk space allocated to relative record files. Any clearing must be done by the application program. Some language runtimes handle this function.
- The system does not support deleting records from relative record files. Any application that must handle deleted records must do so explicitly by using some flag value written in records to be considered deleted. Some language runtimes handle this function.
- The write end-of-file (WEOF) on a relative record file shortens the end-of-medium. You cannot use the WEOF to lengthen the end-of-medium beyond the space allocated to the file.

3.4.4.3 Constraints on Key Indexed Files. The system uses the logical record length you specify when you create a key indexed file for the following purposes:

- Computation of initial and secondary allocations for the file
- Returning the logical record length to programs via Read Characteristics and Open SVCs

The system computes the initial file size from the larger of the initial allocation and the maximum size that you specify. The system also uses the logical record size, physical record size, and allocation to compute the disk space required. Only the disk space for data is computed and allocated. Since index blocks are also taken from the allocated disk space when the file is actually loaded, the file appears to grow when in fact insufficient area was initially allocated. To compensate, specify an initial allocation large enough to accommodate the computed index block space as well.

System handling of key indexed files is as follows:

- The logical record may be any length that fits in the physical record. The KIF system maintains some index information in each block, limiting the actual record size that can be written to somewhat less than the size of the physical record.

- Logical records do not span physical record boundaries, allowing wasted space inside physical records to occur. Correct blocking choice will minimize the amount of waste. KIFs are automatically blocked by the system. (Blocking is discussed in paragraph 3.4.6.)

- The system makes no check that a logical record being written matches the created logical record length. Any regulation of record length must be done by the application program or its runtime.

- The DX10 file system initializes only the log blocks and b-tree roots; no other storage is initialized for KIFs.

- KIFs are always forced write, unless modified by the Modify KIF Logging (MKL) SCI command (described in Volume II).

3.4.4.4 Choosing Logical Record Length. There are certain formulas for determining an efficient number of logical records per physical record, and is essentially the *blocking* concept discussed along with blocking and blocking buffers. Consider the following points when choosing logical record length:

- Sequential file logical records can be any number of bytes, including zero.

- Key indexed file logical records must be an even number of bytes and cannot be zero.

- Relative record file logical records must be an even number of bytes, must all be the same length, and cannot be zero.

3.4.5 Physical Records

Physical records are units of space that correspond in size to the size of I/O transfer buffers allocated by the system. The allocation is based on the value you specify as the physical record length when you create the file. If you do not specify a physical record length, the system generates a default value. Usually, physical records contain more than one logical record. Therefore, they are often viewed as blocks of logical records.

Physical records make disk file I/O more efficient by reducing the number of disk I/O events required for read, write and rewrite operations. Whenever a specific logical record is requested by a program, the entire physical record in which that logical record resides is read into a blocking buffer in memory. As other logical records not contained in that physical record are requested, the physical records containing them are also read into memory, as space allows.

Subsequent requests for logical records often reference records already in blocking buffers in memory. This minimizes disk I/O, particularly for sequential files or sequential access to relative record files. Write operations also occur to these memory images of the physical records, unless *forced* write is specified, as mentioned earlier. As other tasks require memory, the system writes these buffers out to the disk file as a block of data. This is why the practice of placing several logical records into a single physical record is called blocking. Blocking is discussed in more detail later in this section.

3.4.5.1 Choosing Physical Record Size. A file always begins on an ADU boundary. DX10 allocates ADUs to a file, and places the first physical record of a file at the beginning of the first sector of the first ADU. DX10 then places the rest of the records into the file according to the following ADU constraints:

- A physical record always starts on a sector boundary within the ADU.
- A physical record never extends over an ADU boundary unless it starts on an ADU boundary.

For efficient disk file space utilization, choose the physical record size based on the following criteria:

- The physical record size should be a multiple of the sector size.
- The physical record size should be a multiple of the ADU size if larger than one ADU, and should divide evenly into the ADU size if smaller than one ADU.
- If the file is a relative record file, the physical record size should be an integer multiple of the logical record size.
- The physical record size should be about three times the logical record length to take advantage of the blocking feature.
- The physical record size should not exceed 3000 bytes unless all related criteria are met as outlined in the system generation discussion of the memory-resident buffer in Volume V.

Several additional criteria must be considered if files are to be transported between disks with different sector sizes, for example, between a DS10 (sector size of 288 bytes, ADU size of one sector), and a CD1400/96 (sector size 256 bytes, ADU size of six sectors). The amount of wasted space, the time to copy the file using one of the DX10 SCI copy commands, and the size of the file need to be considered. Using the preceding criteria, you can develop a file size strategy for your application that will minimize transport time and trouble. (Refer to Volume II for information on how DX10 SCI copy commands preserve physical record length.)

3.4.5.2 Default Physical Record Size. The default physical record size is used by the system when you create a file and specify zero or a null response for the physical record size parameter. The default is obtained by the system by examining the following parameters:

1. The system inspects the directory in which the file is catalogued. If a default physical record size was specified for it, the system uses that size.
2. If no default record size was specified, the value specified during system generation for the disk drive involved is used. Intermediate directories are not examined.
3. When a disk is initialized (INV) and no value is specified for the default physical record size, the value specified during system generation becomes the default physical record size for the disk's VCATALOG. If a different value is given at INV, that value becomes the default for VCATALOG. The default physical record size determined at INV propagates throughout the disk's directories unless you specify a default physical record size when you create directories (CFDIR).

3.4.6 Blocking and Blocking Buffers

Blocking is the practice of using intermediate buffers in memory to accomplish file read/write operations, rather than by requiring disk access each time file I/O is requested by a program. Blocking makes disk file I/O much more efficient, because each disk event takes a proportionately longer time to complete than the same operation on records in memory.

The block size is the physical record size. The block size represents the size of the buffer transferred between disk and memory.

For blocked files, the file system allocates blocking buffers outside the user's program space out of the same memory area of the machine from which user programs are allocated. DX10 uses these blocking buffers for the disk transfer. (In unblocked files, the transfer is directly to and from the buffer in your task that you specify in the I/O SVC.)

When a program writes to a blocked file, normally the block is modified in memory and is not immediately written to disk. This saves system overhead by minimizing disk I/O operations. (Refer to the discussion of immediate and forced write program attributes for additional information.)

DX10 blocks records on disk at two different levels. Logical records are blocked into physical records, and physical records are blocked into ADUs.

You choose the block size when you select a physical record size. Rules for choosing logical and physcial record sizes are discussed in preceding paragraphs. Formulas for selecting the number of logical records in a physical record are given in the following paragraphs, and must be considered within the context of the rules for choosing record size.

3.4.6.1 Choosing Logical Records per Physical Record. To select an appropriate ratio of logical records to a physical record, you should estimate the required logical record length for the application. Then you can select a physical record size that will be the block size for that file, resulting in the least amount of wasted space. Another factor is the tradeoffs involved in the disk access overhead.

The number of logical records per physical record is called the *blocking factor*. A good blocking factor depends on the choice of physical record length. The best blocking factor is the one that results in the least wasted space (unless disk access overhead is an overriding consideration).

One method of choosing a physical record length is to substitute estimated physical record lengths into a formula. The physical record length that results in the best blocking factor (least wasted space) is the best length. The formulas for determining a blocking factor are different for each type of file. Table 3-4 gives the formulas. They are explained in the paragraphs following the table.

Table 3-4. Blocking Logical Records for Sequential Files

FR = Blocking Factor per Physical Record

PRL = Physical Record Length

LRL = Logical Record Length

Sequential Files:	FR = (PRL − 2) / (LRL + 4)
Relative Record Files:	FR = (PRL) / (LRL), (discard remainder)
KIFs:	FR = (PRL − 16) / (LRL + 6), (discard remainder)

WR = Wasted Space per Physical Record

Sequential Files:	WR = 0
Relative Record Files:	WR = PRL − (FR × LRL)
KIFs:	WR = (PRL − 16) − FR × (LRL + 6)

For sequential files, the system automatically splits logical records over physical record boundaries. When a logical record does not fit evenly into the space remaining in a physical record, it is carried into the next physical record. That is why there is no wasted space for sequential files.

To get the blocking factor for relative record files, divide the projected physical record length by the logical record length, and discard any remainder. You can determine the number of bytes wasted by choosing that physical record length by multiplying the resulting blocking factor by the logical record length, and subtracting the product from the physical record length. This waste represents the space unused for *each* physical record in the file.

To get the blocking factor for KIFs, add 6 to the logical record length, and subtract 16 from the projected physical record length. Then divide the resulting logical record length into the resulting physical record length, discarding any remainder. To calculate the waste in this formula, multiply the divisor (LRL + 6) by the resulting blocking factor, and subtract that value from the dividend (PRL–16). The result represents the space unused for *each* physical record in the file.

You can generally determine the blocking factor and wasted space if you know the physical record size and the ADU size for the disk. You can use the following formula. In the formula, FA represents the blocking factor per ADU, and WA represents the wasted space per ADU.

- If the block size is smaller than the ADU size:
  - Round the PRL up to the next multiple of the sector size
  - FA = ADU / PRL, discard any remainder
  - WA = ADU - (FA × PRL)
- If the block size is larger than the ADU size, the number of ADUs used for each block is:
  - A = PRL/ADU, if a nonzero remainder results, add one to A
  - WA = (A × ADU) – PRL

The total waste for a file is the sum of the wasted space per physical record times the number of physical records and the wasted space per ADU times the number of ADUs in use.

3.4.7 Unblocked Files

Only relative record files can be unblocked files. To specify a file as unblocked, make the physical record length and the logical record length the same. Special-purpose relative record files such as program files, image files, and directory files are automatically created with the physical and logical record sizes the same. (Program files and image files are also created with a physical record length the same as the disk sector size.) I/O operations to unblocked files do not use intermediate blocking buffers. A file management disk I/O routine transfers the requested record directly between the disk file and the requesting task's data buffer.

3.4.8 How DX10 Allocates Disk File Space

DX10 allocates file space on disk on the basis of the following:

- Physical and logical record sizes
- Initial and secondary file allocation parameters
- Whether or not the file is expandable

You specify these parameters when you create the file. DX10 allocates space for the file in multiples of ADUs. The total file space can be as small as one ADU, or as large as the total available space on the disk. If you elect the file to be expandable, the file can grow beyond the primary allocation.

The system allocates disk space to files when a program executes a create file supervisor call and when a program writes sufficient data to a file to fill all previously allocated disk space. When a file is created, FUTIL requests space and builds the file descriptor records to allow a user program to use the newly created file. When a file is expanded, FILMGR requests space and updates the file descriptor record allocation tables to add the newly allocated disk space so the user program can use it.

DSKMGR scans the disk tables (using a first-fit algorithm) for the appropriate disk. Then it locates an area on disk that is large enough for the request, or it locates the largest area on the disk. DSKMGR sets the bits in the bit maps on track 0 (or clears them, for deletes) to mark the correct state of the corresponding ADUs.

3.4.8.1 Secondary Allocation Algorithm. The file expansion algorithm allows a file to grow as needed, and keeps the number of secondary allocations needed to a minimum. The algorithm is intended to keep files from being more than an average of 25 percent empty. (This space can be recovered using the CD command on most systems.)

The secondary allocation parameter indicates the maximum number of records to be placed in the secondary allocation space. The File Utility (FUTIL) uses the secondary allocation value and the logical record size to compute the number of ADUs required to contain records specified in the secondary allocation. The number is never less than one ADU. The system uses this value when the file expands. The system computes the amount of disk space needed to expand a file using an algorithm that involves scaling the number of times the file has been expanded into a multiplying factor by using a modified exponential algorithm. Thus, subsequent secondary allocations automatically and progressively increase in size over the previous allocation.

Files add secondary allocations up to a maximum of 16 secondary allocations. If the first secondary allocation has size N bytes, then the second secondary allocation has size 2N bytes, the third has size 4N bytes, and the sixteenth allocation has 2^{15}N or 32,768N bytes.

3.4.8.2 Exception to the Algorithm. The exception to this algorithm occurs when the available space on the target disk is in smaller segments than the amount of space indicated by the algorithm. This is called *fragmentation*. When fragmentation occurs, the fragments are allocated instead of the indicated secondary allocation. This situation does not allow for maximum expansion of the file because each allocation to a fragment of space still occupies one of the 16 available entries in the secondary allocation table for each file.

The number of expansions is not always the same as the number of secondary allocation entries occupied. When the system extends a file, it first attempts to obtain storage that is physically contiguous to the last secondary allocation (or initial allocation if there are no secondary allocations). If successful, the number of expansions is incremented but the storage is appended to an existing allocation, so an extra expansion slot is not used.

File usage should be arranged so that expansion characteristics do not create a problem. For example, if two files are expanding in a "hopscotch" manner on a disk, they can quickly run out of secondary allocations because each prevents the other from appending extensions to previous allocations. On a freshly initialized disk, hopscotching may not be a problem because the expansions get larger fast enough to fill a disk before the secondary allocation table fills up, rendering the file no longer expandable. On a disk with some fragmentation already on it, the size of available space limits the expansion of the file, and continued expansion can cause the secondary allocation table to fill up.

3.5 LOGICAL UNIT NUMBERS (LUNOs) AND DEVICES

All I/O performed under DX10 is done to logical units rather than devices. This characteristic allows an application program to view different devices equally, and not require special coding for each different device. You can assign a logical unit number (LUNO) to a device or file either in a batch stream or interactively, with the program accessing the device by that assigned LUNO.

This method facilitates what is known as device independent I/O. DX10 also supports device dependent I/O, which allows you to take advantage of particular features of a device. This usually requires an SVC within an assembly language routine. Device dependent and independent I/O are discussed in Section 9.

The following devices can be considered equivalent by a program:

- 911, 913, and 931 video display terminals
- 733 ASR/KSR data terminals
- 703/707 and 743 KSR data terminals
- 820 RO KSR and 781 RO data terminals
- 915 remote video terminal
- 783/785/787 data terminals
- 763/765 bubble memory terminals
- 804 card readers
- 588, 810, 850, 855, 2230, 2260, LQ45, and 306 line printers
- 940 electronic video terminal/Business System terminal
- 840 RO data terminals

This list can be expanded to include any other device that has a device service routine (DSR) implemented as described in the *DX10 Operating System Systems Programming Guide* (Volume V). All physical devices associated with your configuration must be defined to the system during system generation.

3.5.1 I/O Device Access Through LUNOs

When you write a program that performs I/O, you must include the following steps:

1. Assign a logical unit number (LUNO) to the device.
2. Open the logical unit for I/O.
3. Transfer data.
4. Close the logical unit.
5. Release the logical unit.

DX10 maintains a list of logical unit numbers (LUNOs) that indicate the corresponding physical device. LUNOs may be assigned by operator action or by program action and may have one of three scopes:

- Global LUNOs are defined (and available) for all tasks.
- Station LUNOs are defined (and available) for all tasks assigned to a given station.
- Task LUNOs are defined only for the task that defines them.

When you allow the program to assign the LUNO, use an Assign Luno SVC (or equivalent) rather than coding a LUNO assignment into the program. This allows the system to control LUNO assignment. There are two strategies for logical device assignment implemented by selecting the scope of LUNO assignment:

3.5.1.1 Global or Station Local LUNO Strategy. The global LUNO strategy isolates logical device assignment from program execution. For example, an analog to digital (A/D) converter can be used as a process monitor, requiring many programs to interact in the process. It would be convenient to assign a single global LUNO to the A/D converter. All the programs associated with this process monitor could access the A/D data without making provision in each program for logical device assignment.

Station local LUNOs may be used to isolate logical device assignment from program execution in a similar manner. Extending the preceding example, suppose that there are several operator stations, each with an operator terminal and A/D converter. The strategy is similar to the global LUNO strategy: each station can have several programs that access the A/D converter. Thus, it would be convenient to assign a single station local LUNO to the A/D converter. In this case, any program entered at an operator station could access the A/D converter assigned to the station without program provision for logical device assignment. In addition, if the station LUNO numbers are the same for each station, a particular program may be replicated for every station without reprogramming.

3.5.1.2 Task Local LUNO Strategy. This strategy binds resource allocation assignment to task execution and is appropriate for tasks having I/O access to several different devices or files. For example, a language processor (such as a FORTRAN compiler) usually allows the user to specify, on each execution, the input source file (or device), the output object file (or device), and the listing file (or device).

The task local LUNO strategy is appropriate for the preceding example. You supply access pathnames to the processor and have the processor use task local LUNO assignments for logical device assignment. This way, several instances of a processor may be executing at the same time, each instance having distinct I/O channels.

In some applications, it may be desirable to use both strategies; a single program may use global, station local, and task local LUNOs.

The I/O process paths for the two strategies are illustrated in Figure 3-1.

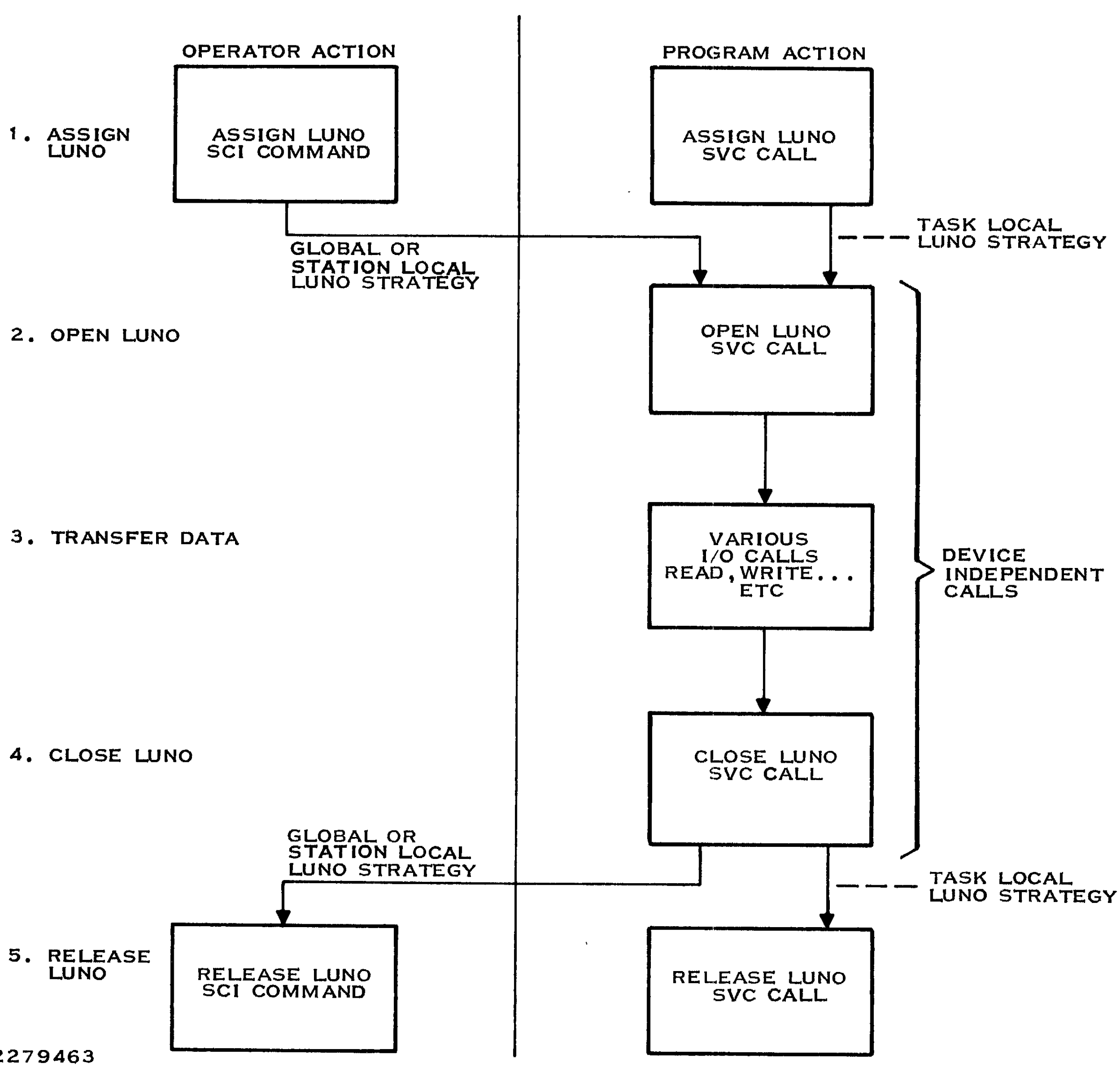


Figure 3-1. I/O Process Paths

CAUTION

When DX10 searches the LUNO tables, it first looks for task local LUNOs, then for station local LUNOs, and finally for global LUNOs. Thus, a task local LUNO may mask a station local or global LUNO with the same number. The system uses global LUNOs in the range >00 to >90 and >D0 to >FF. Use of task local or station LUNOs in this range may produce unpredictable system performance. Typically, the language processors fail. If automatic LUNO generation is specified, there will be no masking problem. The global LUNOs in the range >91 to >CF are reserved for user files.

3.5.2 Access Logic for Devices

DX10 devices are either file-oriented or record-oriented. All terminals and all disk drives (for direct disk I/O) are record-oriented devices. For other devices, record/file orientation is an option selectable during system generation. As described in the following paragraphs, device orientation determines the programming necessary for access control.

3.5.2.1 Record-Oriented Devices. Record-oriented devices may be shared at the logical record level. Any task may access a record-oriented device through any LUNO assigned to the device. The LUNO must have the appropriate scope. A task accessing a record-oriented device need not OPEN the associated LUNO. If the program can execute in an environment where it accesses a file-oriented device, the open call must be coded to maintain device independence.

3.5.2.2 File-Oriented Devices. Only one task at a time may have access to a file-oriented device. A task gains access to a file-oriented device through the OPEN call. Before any other task may OPEN a file-oriented device, the previous user must have CLOSED it. The following rules apply to file-oriented devices:

- Any number of LUNOs may be assigned to a file-oriented device, but only one may be open.
- If an attempt is made to open a LUNO which is assigned to a file-oriented device which already has an open LUNO, the attempt is rejected with error code >0F.
- If an attempt is made to open a LUNO which is assigned to a file-oriented device and if the LUNO is already open to the same task making the attempt, then the OPEN is processed and no error occurs. This provides a convenient way to modify attributes assigned by the OPEN call.

The system automatically closes all open LUNOs when the task terminates.

For information on using LUNOs in high-level languages, refer to the appropriate language programmer's guide. For assembly language programs, you must use SVCs to access LUNOs.

3.5.3 Using the Printer and Printer Files

The printer on a DX10 system can be accessed in several ways. The printer can also have one of two access modes (record oriented or file oriented) depending on a system generation option. (Refer to paragraph 3.5.2.) This paragraph describes the format of data that you use to access a printer on a DX10 system.

All I/O, as described earlier, is accomplished by using SVCs that specify a LUNO assigned to the desired I/O device or file. Most high-level languages have a method of specifying the connection between the I/O unit on the language and the access name of the I/O unit of the DX10 system. The language runtime assigns the LUNOs and opens them.

The printer is accessed by one of several methods in DX10:

- Direct writes to the printer — This is accomplished by assigning a LUNO to the printer device and using write SVCs to write to the LUNO. In this mode, the user program must supply all carriage control to delimit lines and pages. The user program can also supply special control codes for special printers (for example, letter-quality printers). The DX10 program that services printer requests does not remove any characters from the buffers that the program requests to be written.

- Writes to a disk file, automatically queued for printing — This is accomplished by assigning a LUNO to a printer pseudo-device, LP$x, where x can range from 1 to 9. (LP$1 prints on LP01, LP$2 prints on LP02, and so on.) DX10 creates an intermediate file and assigns a LUNO to that file. The program's writes then go to the file. When the LUNO is released, DX10 queues the file for printing through the SCI print queuing capability. (It simulates a Print File (PF) command through the background of station zero.) The format of the file follows the rules for printed files (as described in the next item). It is not possible to specify the equivalent of the PF command's ANSI format using the LP$x devices, or to specify the number of lines per page. Your system disk must also have the directory .S$PRINT on it.

- Writes to a disk file — To accomplish this, the program assigns a LUNO to a disk file and writes the records to it. Then, the user or an SCI command procedure uses the PF command to queue the file for printing. A program associated with SCI then assigns LUNOs to the file and the printer, and copies the file to the printer device.

The PF command handles the file differently depending on whether or not it finds any non-ASCII characters (character code less than >20) in the first record. If the command finds any such codes, which include special control codes for printers, it assumes that the whole file contains all necessary carriage control and the records are copied from the file to the printer with no changes. If the PF command finds no carriage control characters, it causes a form feed before printing, adds a carriage return and line feed to each line, and inserts a form feed after the number of lines specified in the lines per page prompt. Therefore, you should not concatenate files together for printing when some have their own control characters and others do not.

If you specify 255 lines per page when using the PF command, it adds a carriage return and a line feed to each line. It does not add any other control to the data as it is printed.

Text Editor files do not have any carriage control characters in them. The outputs of programs such as compilers, the Link Editor, and the SCI batch stream process do have such characters. If you need to determine whether a file has such characters, use the List Logical Record (LLR) command to list the first record. Inspect the record for any character code less than >20.

4

Designing Applications for Data Protection

4.1 INTRODUCTION

Successful applications must be more than fast and efficient. They must provide maximum protection for the user's data. There are two major aspects of data protection, as follows:

- Consistent and accurate error reporting
- Transaction logging

Undetected errors can proliferate throughout a user data base, causing inaccurate reporting. Detecting errors, capturing a meaningful error code, and returning the error code to the program or operator prevent these errors from proliferating.

Transaction logging provides a record of changes to data, which can then be reconstructed in the event the data storage media is lost or damaged.

The responsibility for effective error detection and transaction logging falls primarily on the programs that actually massage the data. Additionally, log files, recovery programs, and a good recovery scenario support the data protection effort.

This section discusses error reporting and transaction logging principles. Since regular data backups are an integral part of data protection, also refer to Volume II for a general discussion of backup and recovery techniques, and backup/recovery SCI commands.

4.2 ERROR REPORTING

Most errors affecting data operations occur when a supervisor call (SVC) is being executed. DX10 reports any error codes occurring when an SVC call is issued from an assembly language program. These error codes and recovery suggestions are documented in Volume VI.

However, you cannot get to the error code if the SVC is executed through the syntax of a high-level language. Some high-level languages have status subroutines to retrieve these error codes, but often the languages do not report error codes directly.

When using high-level languages, it may be necessary to include subroutines to obtain those error codes. For any error significant enough to terminate program execution, the program should report both the DX10 error code, and the activity in the program at the time of the error (such as which file is being accessed).

4.3 TRANSACTION LOGGING PRINCIPLES

Transaction logging refers to the practice of retaining a description of all changes made to a collection of data. In the event of major software or hardware failure, you can use transaction logging to restore your data, with the possible exception of the transaction in progress at the time of failure. To restore the data, execute a special program against your record of all transactions. The program should recreate each transaction that occurred since the last backup, and therefore update the data base.

In this discussion, the description of changes is called a *transaction log*, and is written either on magnetic tape or disk file. The collection of data is usually called the *data base*, and is used here to mean all user data rather than only files maintained by the TI DBMS system. The data base can consist of any of the following:

- Key indexed files
- Relative record files
- Sequential files
- DBMS files
- Information maintained in the memories of special-purpose devices connected to the DX10 system

The program used to read the log and reproduce the changes to the data base is called the *recovery program*, and the process of restoring from backup and running the recovery program is called *recovery* or the *recovery procedure*.

4.3.1 When to Use Transaction Logging

The costs of implementing and maintaining a transaction logging system may not outweigh the benefits. The primary motive for using transaction logging is to possess the capability of restoring such data to operational integrity in as short a time as possible after system or media failure occurs. If your data base does not contain critical or volatile data, or your volume of transactions is relatively low, a regular data backup program may be sufficient. Any time data is lost, reenter the transactions manually.

The TI DBMS system includes a transaction logging subsystem. If you elect to use it, you may not need to read this section. The principles outlined here are used in the DBMS subsystem.

If you elect to implement a transaction logging system, you should be familiar with the data flow of your applications. Successful implementation of a transaction logging system depends heavily on the planning and designing stages.

4.3.1.1 Cost of Implementation. The following list contains some component costs of a transaction logging system:

- Implementing the additional code in the application programs to create the log entries
- Implementing the recovery program
- Maintaining the extra device and/or online storage capacity necessary to write the transaction log
- Extra operator effort to maintain the log device and its media (for example, changing tape reels when one fills up)

The cost of restoring data (by whatever means) has two basic components:

- Time required by data entry operators to rekey the lost data, or the time required by the computer operator to run a recovery procedure
- Time lost by the users of the system while waiting for the data to be restored

Depending on the use of the data, the incidental and consequential costs of a failure and/or the time to recover can be high, and you may want to include such costs in your evaluation of the potential benefit.

Generally, use transaction logging whenever the overall cost of using it is lower than other methods of protecting your data base integrity.

4.3.1.2 Computing Mean Time Between Failures. A comparison of absolute costs can be difficult to compute however. For one thing you must also determine the mean time between failure (MTBF). A comparison of the relative cost of restoring data with different methods can be made without knowing the MTBF of the system, but it does not allow you to compute the payback on initial development or hardware costs.

If you have an extremely critical data base the incidental and consequential costs of a failure can be high enough to make MTBF a very small factor. For example, a hospital data base maintaining prescription data for its patients can have an extremely high consequential cost if the data is not up to date and readily available. In such a case, the shorter recovery time furnished by transaction logging may be preferred over a less costly manual recovery system.

The MTBF of a system is composed of three factors:

- The overall MTBF of the system hardware. TI Field Service can sometimes provide this data.
- The overall MTBF of the system software. Software failures can induce inconsistencies in a data base. Table area overflow is an example.
- The MTBF of operational procedures. Operator error including backing up to the wrong disk pack, deleting necessary files, restoring from the wrong backup, and improperly archiving logs are examples. Proper design and documentation of procedures guards against many of these. You can gain additional security by delete-protecting files.

To arrive at the MTBF, first determine the factor with the longest MTBF. Then note how many instances of the other two factors occur during that time. The overall MTBF is the time interval divided by the total number of possible failures.

4.3.1.3 The Log File as Audit Trail. The log file can serve as an audit trail, but the following differences should be noted:

- A transaction log does not need to contain inquiries, while an audit trail can require a record of all accesses to the file, including inquiries. Available storage medium may prove to be inadequate for such a volume of data.
- A transaction log does not need to contain the identification of the user who initiated the change, while an audit trail may require user ID or other information.
- A transaction log must be written to a different device from the one containing the data base. Ideally, it should be written to a device on a different hardware controller. Otherwise, you risk losing the transaction log along with the data base when failure occurs.
- The log must contain enough data so that the recovery program can remake all the changes to update a copy of the file (as restored from backup) to the point where the failure occurred. Such data would include:
  - File position data such as key values or record numbers. Do not use internal pointers such as currency blocks for KIFs since these are not retained when the file is backed up with usual backup utiities.
  - Type of change activity (for example, delete, create, change).
  - Some identification of which fields of the affected record changed.
  - The new data, either new field contents or amount of change of the value in a numeric field.

4.3.2 Designing the Recovery Procedures

The success of any recovery operation depends on the design of the recovery procedures. The following discussion presents a possible scenario for data recovery. First, however, some terms must be defined.

The term *transaction* usually refers to any changes made to a data base that accomplish a given function. For example, entry of an order may require changing quantity in stock in one file and creating an invoice in another file. Changing the quantity and producing the invoice comprise one transaction, although changes to many records in several files may be involved. The word *transaction* in this section refers to a function which when properly completed leaves the data base in a consistent state. If it is not completed, the data base is in an inconsistent state (for example, quantity in stock may be out of balance). Logging is used to help you restore a consistent state, although logging by itself may not be enough as noted below.

Each individual record written in the log is referred to as an *item* or *entry* in the log. More than one log entry may be necessary to describe one transaction.

The term *transaction level logging* refers specifically to the case where more than one entry is made in the log for one transaction, and sufficient control is exercised to guarantee that for each transaction, the recovery program is able determine which log entries comprise a complete transaction and thus avoid making some of the changes related to the partially-completed last transaction.

Without this control, restoring a data base using a transaction log may finish the restore process by making only part of the changes related to the last transaction. The extent of the changes depends on when the failure interrupts the transaction or the logging process. This last transaction can require close consideration to determine exactly which files have been updated and which have not. In a very complex application, a highly customized procedure may be required to complete the recovery operation for this last transaction.

The general recovery scenario is given in the following steps. Considerations for designing specific steps follow the scenario.

1. Back up the file(s) using a suitable utility.
2. Continue normal activity for the usual time interval until the next backup. This produces a transaction log of the activity on the file.
3. If all goes normally, archive the transaction log, start a new log, and then continue with step 1. If a failure occurs during the use of the file, proceed with step 4.
4. Restore the file from backup.
5. Run the log recovery program to bring the file up to date to the time of failure.
6. Go to step 2 and proceed with file activity.

4.3.2.1 Step 1: Backup. The utility suitable for backup depends on the type of file position data retained in the log file. If your position data consists of internal pointers, such as KIF currency blocks, you must use a form of backup that retains the physical record contents intact. TI does not recommend using this form of backup, particularly with KIFs, because KIF currency blocks are not well enough controlled to provide a consistent means of recording the position in the file at which the update should be made. (DBMS keeps its own internal pointers in relative record files, and exercises its own control to assure that the log entries are usable. Therefore, DBMS only requires that the logical records of the DX10 file remain intact.) The record number for sequential and relative record files is sufficiently controlled, and does not require a form of backup that retains the physical records intact.

Additional factors in choosing a suitable backup utility are its speed and its ability to verify the copy. Verification is very important, and should always be performed when backing up a file. Speed is usually relative to the size of the file, the speed of the backup medium, and requirements for availability of the system during the backup process.

4.3.2.2 Step 2: Activity Interval. Set the time interval between backups based on the following factors:

- The volume of transactions processed in a given interval
- The length of the supporting log entries
- The capacity of the log file medium
- The time required to back up the data base files
- The time it takes to perform a restore operation. The longer the log file, the longer it takes to perform a recovery.

If the time to recover exceeds the time allowed for a recovery based on user requirements and/or cost, consider making backups more often to shorten the log file.

4.3.2.3 Step 3: Archive the Transaction Log. Archiving the transaction log means making a copy of the log on another disk pack or magnetic tape, and saving it offline where it can be available if needed. The archive medium usually depends on the relative cost of the device and/or medium. A small transaction log may be effectively archived on disk, but you may want to archive a larger log on magnetic tape. You should archive a log each time you make a backup and start a new log. In addition, keep accurate records so you can easily determine which logs to use if a recovery operation needs to be done. The record should be handwritten because such records kept on disk files are subject to the same kind of failures that can damage your data base.

The number of archived logs you keep will depend on how many generations of backup you need to keep. Note that in a three-disk rotation system described for general backup, after making a backup you have two identical disks and one "old" backup. Thus, the log made by operations during the previous interval is all you need to save. As you process data, you will accumulate a current log. These two logs are used in recovery.

4.3.2.4 Step 4: Recovery. Usually, only the current disk or one or two files on the current disk are lost. In such a case, restore the files from the backup made at the beginning of the activity interval and execute recovery against the log made during the current operations (since the backup was taken). You must always use exactly the log entries made since the backup was taken.

A three-disk rotation system allows recovery if the backup made at the beginning of the activity interval is unusable. Restore from the previous interval's backup disk and run the recovery program with both the archived log and the current log. Plan to archive enough logs so that the oldest backup in the rotation cycle can be used to restore the data.

Suppose, in a three-disk rotation, that while you are making a backup of your current disk a hardware failure damages it beyond use. Now your current disk is unusable, and the copy disk is also unusable because it was only partly done. You now have only one backup. It is unwise to try to accomplish recovery without first determining the cause of the preceding failures. You could not afford to damage your last backup.

If you want to further protect against loss of that backup (made at the beginning of the previous interval), your rotation system can contain four disk packs (or three sets of tape). Archive two logs to use along with the current log.

Note that a four-disk rotation system makes it possible to omit the copy operation in step 4 used to restore the file(s) to the state it had at the beginning of the current interval. It is only necessary, if a full disk backup is used, to mount the backup pack and apply the logs to it. However, a four-disk rotation allows you to do that only once; after running the recovery you have the equivalent of a three disk rotation backup. If, in recovery (after omitting step 4), you have to restore from backup again, you should copy the data to a backup to avoid the possibility of a complete loss.

4.3.3 Blocking Log Entries

Log entries can be blocked by the application program so that one write operation to the log file will write several entries. This can be done with either a disk file or tape file log, but it is usually only done with a tape file. This is because DX10 does not handle blocking tape records but does handle blocking file records. The principal problem with blocked log entries is ensuring that the log entries can be applied to the data base by the recovery program to produce the correct state of the data base.

Your log must retain the order of the updates to ensure that the data is correct after recovery. If your log contains updates in a different order than the chronological order they were applied to the original file, log entries should specifiy changes rather than the result of a change, or should contain a tag or ID so the recover program can order them properly.

The following example illustrates the possible faulty results from such a design. In this example, assume there are multiple users on the system, and you are logging transactions on an inventory master file, with each program blocking its own updates. At the beginning of the example, Item X has 4 units in stock, and Item Y has 10.

1. User A removes 1 Item X. The new quantity recorded in the data base and log block is 3.
2. User B removes 1 Item X. The new quantity recorded in the data base and log block is 2.
3. User B removes 1 Item Y. The new quantity recorded in the data base and log block is 9.
4. User B's block is full; write it to the log file.
5. User A removes 2 Item Y. The new quantity recorded in the data base and log block is 7.
6. User A's block is full; write it to the log file.

The log contains the following entries:

Item X new QTY 2.
Item Y new QTY 9.
Item X new QTY 3.
Item Y new QTY 7.

If you have to recover from the log, the quantity for Item Y would be correct but that for Item X would not.

The operation illustrated in the following example avoids the problem. Again, assume Item X begins with 4 units in stock and Item Y begins with 10:

1. User A removes 1 Item X. New quantity recorded in the data base is 3; the log block shows a change of −1.
2. User B removes 1 Item X. New quantity recorded in the data base is 2; the log block shows a change of −1.
3. User B removes 1 Item Y. New quantity recorded in the data base is 9; the log block shows a change of −1.
4. Block is full; write it to the log file.
5. User A removes 2 Item Y. New quantity recorded in the data base is 7; the log block shows a change of −2.
6. Block is full; write it to the log file.

Now, the log contains the following entries:

Item X decrease by 1.
Item Y decrease by 1.
Item X decrease by 1.
Item Y decrease by 2.

If you need to recover from the log, all quantities will be correct using this procedure.

If you have data in a form that cannot be readily described with a change you must write the entire new record in the log instead of just the changes. You must also guarantee that the log entries are applied to the data base in the correct order. One way to do this is by locking the log file before beginning the update to the data base. You can write whatever log entries you need, block them to tape if necessary, and unlock the log file after the transaction is complete.

The recovery programs should be coded in such a way that no user input is needed after the data base update process begins until after it is finished.

You can use tags or some other ID that allows the recovery program to order updates correctly. The recovery program must be able to read and save several blocks or entries, and determine when it has read a complete transaction before applying that transaction. You can use a separate entry in the log marking the end of the transaction, or you can place a flag in the last entry of a transaction. The application programs may have to use a shared record to assign transaction IDs. This shared record should be handled with locking, in much the same way as the flag record described for relative record files. Generally, the transaction ID will not be in chronological order. Only a time and date stamp made in the correct way will have the chronological order necessary to allow proper recovery from tape. The time and date stamp concept is discussed in the following paragraphs.

4.3.4 File Type for Transaction Log

The preferred file type for a log is sequential. A magnetic tape is a sequential file media, but log files can also reside on disk files. If you use a magnetic tape, be sure to note the special considerations for tape files discussed later.

If log entries have a significant number of blanks, you may elect to make it blank suppressed. Whether or not you make it forced write depends on how you will access and control the file.

Its physical record size need not be large, especially if forced write is selected. Since the forced write attribute makes the system write the blocks to disk any time a record is placed in them, a large physical record size can be detrimental to system throughput.

If several programs will be updating the log, a file type that will support shared write operations (relative record file or keyed index file) could be used, but you must ensure that each program uses the correct records. You can select either the first record or a record of another file to be used as a flag record, and have each program use the following algorithm to gain access to the log file:

1. Attempt to read the flag record with lock.

2. If it cannot be locked, delay a short time and go back to step 1. Some languages will perform the delay and retry function for you.

3. Update the data base and write the desired log entries.

4. Unlock the flag record. If the flag record contains next record information, use a rewrite to update that information.

This makes transaction level logging easy. You can depend on the read file characteristics operation to yield the correct end of file record number only if you can guarantee no other program can write to the file between the time you read characteristics and finish writing. Locking a flag record guarantees this condition.

Several programs can use a sequential file, but each program must open the file and close it for each log operation, according to the following algorithm:

1. Attempt to open the file with exclusive write or exclusive all access privileges. An open extend operation will cause the program to write log entries at the correct place. It is not necessary to use a flag record with a sequential file or tape.

2. If you get an access conflict error, delay a short time and go to step 1.

3. Update the data base and write entries to the log as needed.

4. Close the file.

Transaction logging is easy if a file without forced write is used, because the close operation causes all records to be written and the end of the file to be updated. With a forced write file, the records are always written and the end of file is always kept up to date without a close operation, which requires a flag in the last entry for a transaction to achieve transaction level logging.

It is desirable to put a time and date stamp in each record of a disk file transaction log. (Use the same algorithm as given below for tape blocks.) It is less necessary than with tape, but it will give you a means of identifying records so that if you have to restore logs in a certain order you do not have to depend on the last update date for the file. Copying the file may alter the last update (depending on the utility used) but not the time and date stamp in the record.

4.3.5 Tape File Considerations

If you use magnetic tape for a transaction log, some special characteristics of tape must be handled by the log and recovery programs. At the time of a system failure, depending on the nature of the failure and the position of the tape, the end of the current log is indicated by either a tape error, an end-of-tape indication, an end-of-file mark on the tape, or by a log entry whose date and time stamp indicates that it was made earlier than the last entry processed. Each block must contain a date and time stamp that reflects the chronological order the blocks are written to the tape.

To correctly stamp a block, the program writing the log should use the following algorithm:

1. Issue a Do Not Suspend SVC of sufficient length to cover program execution up to and including the write operation.
2. Issue a Get Date and Time SVC (which does not terminate the DO NOT SUSPEND status).
3. Place the time and date in the block that is ready to be written.
4. Write the block. This will automatically nullify the effect of the Do Not Suspend SVC.

A tape can be made to behave like a sequential file in terms of how the program gains access, although the error returned by an attempt to open a tape that is open by another program is different from the error returned with a sequential file. (You use the same algorithm.) This is done by making it file oriented during system generation. Alternatively, a tape can be made to appear sharable like a relative record file by making it record oriented during system generation. A flag record is required to control access as described in the paragraph on blocking log entries. A tape has no record number; its physical position determines where the next record will be written. It always behaves like a forced write file.

If you block log entries for tape, allow for the fact that a block being composed in memory at the time of a failure will be lost. If a transaction requires more than one block to write the log entries, a flag will be necessary in the last block so the recovery program can avoid applying a partial transaction at the end of the log. Blocking is generally desirable because a tape has interrecord gaps that can take most of the tape if the records are short, limiting the capacity of a reel to something far less than its theoretical capacity.

A tape being written at the time of a failure has no end-of-medium mark like a file on disk does. A tape reel may be new or it may be mostly filled with a previous log. If a failure occurs when the tape is only partly used, the tape may have no errors on it or it may have a parity, format, or timeout error at the end of the current log, depending on where the tape stopped on the last operation.

The recovery program must be able to stop when it detects any one of several conditions:

- Detection of an EOF mark on the tape
- Detection of end-of-tape status
- A tape error occurs
- Detection of a date out of sequence
- Detection of a disk file EOF mark

The program must be able to detect an EOF mark on tape in case you need to recover with more than one log. The EOF mark indicates the end of a normal log. The program then stops so you can continue the recovery using the next log.

If you have a high volume of transactions, you may require a multi-reel log. The program must be able to execute an orderly stop at the end of the tape to allow you to mount the next reel.

If a tape error occurs, the program should stop immediately. Such an error could be an actual error in the media, preventing recovery of a particular transaction, or it could just indicate the end of the current log. That is, the record producing the error is a record left from some preceding log, and was not completely written over by current logging.

To determine the cause of the error (other than a timeout error), the program should read the time and date stamp of the record following that record producing the error. If the time and date precede the previous record, then you have reached the end of the current log, and recovery is complete. If the time and date of the next block follows that of the last block processed, the program should flag an error condition. The record will require special recovery efforts.

If an error (other than a timeout error) occurs when attempting to read the next record, then the first error could easily be a media error. The transaction log tape must be kept relatively new, and the tape drive should receive regular maintenance to minimize the risk of such an error occurring in the middle of the transaction log.

The system time and date must be maintained as correct at all times so that if the date of the record just read is prior to the date of the last record read from the log tape, it always indicates the end of the current log. If an operator gives the incorrect date and time when executing the IS command, special recovery efforts may be necessary.

The program should be capable of running from a disk file, in case your tape drive experiences hardware failure. If you archive your logs on disk in addition to archiving on tape, you have an optional means of recovery.

5

Programming with Assembly Language on DX10

5.1 INTRODUCTION

This section describes the DX10 assembler, and discusses any special considerations you need to program in TI assembly language.

The 990/99000 Macro Assembler (SDSMAC) is the macro assembler supported by DX10. SDSMAC supports the full set of instructions from the 990 computer inventory, and additionally supports the following capabilities:

- Extensive macro language
- Use of parentheses in expressions
- Logical operators in expressions
- Relational operators in expressions
- Two additional output options
- Workspace pointer directive
- Copy source file directive
- Define operation directive
- Transfer vector pseudo instruction
- Common/Program/Data segment directives

The macro assembler recognizes the TI assembly language instruction set. This instruction set is detailed in the *990/99000 Assembly Language Reference Manual.*

DX10 provides a number of supervisor calls (SVCs) that perform services and functions such as device and file I/O, task synchronization and program support. As described in Section 7, SVCs are accessed from assembly language programs by coding a supervisor call block and then issuing the extended operation (XOP) assembly language instruction. Coding the call block involves placing the SVC code and the data required by the SVC routine into a specific format that can be recognized by the routine. The XOP instruction is issued in the task's procedure area, and contains the label on the first word of the call block, and the XOP level. This level is 15 for pre-written SVCs, 0–14 for user-written SVCs that can be added during system generation. Refer to Volume V for informaton on writing SVC processors.

SVCs can be privileged or nonprivileged. Privileged SVCs are discussed in Volume V, and include such operations as direct disk I/O. Nonprivileged SVCs are discussed by functional category in Sections 8, 9, and 10 of this manual.

The remainder of this section discusses some specific points of assembly language programming, such as how to take advantage of the concept of program segments, and how to transfer control to the task to begin execution. An assembly language programming exercise is included at the end of the section.

5.2 GENERAL PROGRAMMING CONSIDERATIONS

Previous sections describe the general characteristics of the DX10 I/O system and program handling techniques. Although important, much of the material is geared to the general programming audience. The following paragraphs enhance that material and supply information specifically for assembly language programming.

5.2.1 Assembly Language Program Segmentation

Assembly language programs are composed of one or more segments, as described in Section 2. The types of segments possible in an assembly language program are:

- Task segment
- Procedure segment
- Overlay segment

All programs have a task segment, which is the portion of code that is replicated whenever that task ID is bid by the system. A task in DX10 is any activation of a program. Each time a task is bid, it is given a runtime ID for the time it is executing. If the same task is bid several times, each activation executing concurrently has a different runtime ID. The task segment usually contains the program's data, so it is usually not reentrant. Task segments cannot be shared, even if they are reentrant.

Procedure segments are sharable. They are not replicated at each program activation. When a program containing both a task and a procedure segment is bid several times, only the task segment is replicated and assigned separate runtime IDs. Each replication of the task shares the same procedure code. A procedure is identified by its installed ID and the program file it is loaded from.

Overlay segments are loaded over other seldom-used program code in the task segment. They are specified as overlays when you link the program, or by installing them as overlays. Once loaded, a BL or BLWP assembly language instruction accesses the code. During execution, the task segment code manages the overlays.

As discussed in Section 2, sharing procedure segment code saves memory space. It can also save overhead, since procedure segment code is usually not rolled in and out. Therefore, when you design a program, separate the code into functional segments whenever practical. Usually, the decision is based on the relative sizes of the task and procedure and how many copies will be in use for an extended period of time.

To aid in coding shared procedures, you can code the modules using the DSEG directive. DSEG will produce a data part that can be separated by the Link Editor and placed in the task segment, while the procedural code (PSEG) is placed in a procedure segment. (The *Link Editor Reference Manual* discusses DSEG and PSEG in detail.)

Task, procedure and overlay segments must be assembled, linked together, and installed. Paragraph 5.3 discusses these steps.

5.2.2 Attaching Procedures to Tasks

The "attach" process makes sure that the flags in the program file entry for the task indicate that the procedure is to be loaded with the task. This can be done by linking, at the time you install the task (IT), or with the MTE command (Modify Task Entry).

5.2.3 Transfer Vector

DX10 transfers control to a task through a transfer vector in the task segment of program. The first three words of an assembly language program task segment must be the transfer vector, coded as follows:

1st word:	Initial workspace address (WP)
2nd word:	Initial program entry point address (PC)
3rd word:	End action entry point address

An example of the code required is as follows:

```
               IDT 'TASK'
               REF PROC1,ERPROC
               DATA WSPACE
               DATA PROC1
               DATA ERPROC
WSPACE         BSS 32 WORKSPACE
                 *   *
               Task Data
                 *   *
                 *   *
PROC1          EQU $          $PROGRAM ENTRY POINT
                 *   *
                 *   *
ERPROC         EQU $          $END ACTION ENTRY POINT
                 *   *
                 *   *
               END
```

2284641

5.2.4 End Action Routines

An end action routine cleans up after an error. End action routines are user-written, and controlled by the program. They are accessed through a specified address called the end action entry point.

If the end action entry point address is less than or equal to >0F, DX10 terminates the task when a fatal error is detected. If the address is greater than >0F, control is transferred to the specified address. DX10 releases resources from the task and takes it out of execution when one of the following conditions occur:

- A fatal error occurs and no end action routine exists.
- The end action routine executes an End Task SVC (>04).
- The end action routine itself receives an error and has not executed the Reset End Action SVC.

A Reset End Action SVC should be used only if the end action routine causes normal execution to resume.

5.2.5 Using SVCs

Assembly language programs access SVCs through a supervisor call block and the XOP instruction. You must code the SVC call block in the task segment, and issue the XOP instruction from the procedure segment, task segment or an overlay. Since each SVC has specific parameters, each call block is particular to the SVC. All SVC call blocks include the hexadecimal code number unique to that SVC.

SVC call blocks are generally 1 to 14 bytes long, but can be longer if the particular SVC requires more information. SVC call block formats are described in subsequent sections. Section 7 discusses SVCs, call blocks, and general usage, while Sections 8 through 11 supply specific information about each SVC in the following functional categories:

- Program support calls
- Device I/O calls
- File I/O calls

5.2.6 Using Subroutines

You can also implement assembly language subroutines with any assembly language program, by coding the routine, then using a series of user-designed assembly language instructions to access the subroutine. Usually, subroutines are sections of code that perform a specific function. The advantages of using subroutines are:

- The main program can be smaller and more understandable.
- They support top-down, logical structure by providing modularization.
- They lessen programming effort, since a subroutine, coded one time, can be accessed from several different places in the program.

Any subroutine can contain a call to another subroutine. However, certain conventions must be used.

In order to implement a subroutine, you must have a calling sequence, and return logic. A calling sequence is a series of assembly language instructions that save the current place in the main program then transfer control to the subroutine code. The return logic is the set of instructions that retrieve the address of the last instruction executed in the main program and then transfer control back to that place.

There are two types of subroutines:

- Subroutines that use the same workspace as the calling program, therefore using the Branch and Link (BL) instruction
- Subroutines that require a new workspace, therefore using the Branch and Load Workspace Pointer (BLWP) instruction

Information on the specifics of implementing subroutines in assembly language are found in the *Assembly Language Reference Manual.*

5.2.7 Using Overlays

There are two directives used for coding and controlling overlays. These are the REF and DEF. A REF directive to a DEF in an overlay causes the Link Editor to resolve the REF to the location the DEF would be if the overlay were linked completely in the root (phase 0). Refer to the *Link Editor Reference Manual.*

Any number of subroutines can be placed in an overlay. They need not be related in function, although most programmers select related functions. If DSEGS exist in modules linked into an overlay, the data is placed with the code. This means that every time an overlay is loaded, its data is reinitialized.

The Link Editor allows you to select automatic overlay loading. With automatic overlay loading, the Link Editor builds a table in the task segment that allows it to intercept a call to a subroutine in an overlay. It checks if the overlay is loaded, and if not, loads it.

5.3 PROGRAMMING WITH ASSEMBLY LANGUAGE

The following paragraphs describe the steps necessary to successfully write and implement an assembly language program. SCI commands mentioned in this section are discussed in detail in Volume II.

5.3.1 Writing Assembly Language Programs

You can write assembly language programs using the Text Editor (described in Volume IV). You can also implement programs prepared externally by entering them into the system using a card reader or magnetic tape. If the externally-prepared program is in source code format, copy the program into a sequential file and then proceed with assembling and linking, or assemble from the tape or card reader. If the program is already assembled, copy it to a file and link it.

To generate an assembly language program using the Text Editor, you first invoke the editor using the XE command, press the Skip key to have the editor generate a fresh sequential file (take the defaults for exclusive edit and line length prompts). You can then enter the program code line by line. When you finish entering source code, exit the Text Editor using the Quit Edit (QE) command. Respond NO to the ABORT? prompt, and supply a pathname in response to the OUTPUT FILE ACCESS NAME prompt. The editor writes the source code to that file in printable format. The file does not need to be precreated, but if it is, it must be a sequential file. (Refer to the *Text Editor Manual* (Volume IV) for complete information on using the Text Editor.)

NOTE

> Throughout this manual, the names of keys are generic key names. In some cases, the names on the keycaps of the terminals match the generic key names, but in many cases they do not. Appendix A contains a table of key equivalents to identify the specific keys on the terminal you are using. Drawings that show the layout of the keyboard of each type of terminal are also included.

If the program is in segments, such as a task segment with attached procedure segments or overlay code, you can generate the segments separately, and then link them together using the Link Editor.

When generating an assembly language program in one module, take the desired structure into account. If the program is composed of a task segment plus attached procedure segments or overlay segments, and you want to use sequential libraries in the Link Editor, you should generate the code so that the procedures are first, the task is second, and any overlay code is last. This way, when you assemble the program, the object output will be in the correct order required for linking and installing. (If you assemble in separate modules, you can link them in any order.)

5.3.2 Assembling Assembly Language Programs

To assemble the source code, use the Execute Macro Assembler (XMA). You can assemble modules of the program separately, or you can include them in one module.

When executing the XMA command, supply the pathname where you stored the source code as the SOURCE ACCESS NAME. You also supply a pathname for the OBJECT ACCESS NAME, where the assembler places the assembled output. The LISTING ACCESS NAME is a file for the results of the assembling operation, showing the program line-by-line, the address references, and any errors that occur. The ERROR ACCESS NAME is a file where only the error references, if any, are written. Enter the appropriate response to each prompt and press the Return key after each response. The assembler begins execution as soon as all the entries are made.

```
MACRO ASSEMBLY COMPLETE, nnnn ERRORS, nnnn WARNINGS
```

If the assembler detects any errors, you must reenter the Text Editor and correct the source file. Consult the *Assembly Language Reference Manual* if you need help in correcting any of the errors. When all the errors are corrected, reassemble the program.

5.3.3 Linking Assembly Language Programs

The Link Editor links separately-assembled segments into a program. It can also link segments that you assembled together in one module. Assembled task segments that are entirely self-contained (no DSEGs or REFs) do not need to be linked and can be installed directly into a program file using the IT command. Such tasks can have attached procedures, but you must reference data and/or routines without REFs (or DEFs).

If the assembled segment contains references to external programs or modules, you must link the segment to those modules, since the Link Editor resolves external references between two or more independently assembled modules.

The *Link Editor Reference Manual* focuses primarily on how to link assembly language programs. There are several variations on the procedure that depend on the program structure. They are discussed in detail in the manual, and the general steps are outlined in the following paragraphs.

You must create a control file for the Link Editor, as described in the *Link Editor Reference Manual*. The control file specifies the assembled object modules that you want to link together, and contains link directives. You can link tasks and procedures, or produce partially linked modules to be included in other modules.

When you execute the Link Editor, (using the XLE command), supply the control file pathname as the CONTROL ACCESS NAME. Also supply pathnames for the output file and the listing file (generated as the Link Editor executes). You can use the listing file to facilitate debugging. Accept the default values for the print width and page length prompts.

The output of the Link Editor is in one of three formats. You specify the format you want in the control stream. Two of the formats, normal tagged object and compact tagged object, are output to a sequential file and must be installed in a program file before execution. The third format, image, is installed by the Link Editor directly to a user-specified program file. If you specify IMAGE format in the link control file, you do not need to install the program.

5.3.4 Installing Assembly Language Programs

Installing a program means placing it on a program file. As mentioned in Section 2, you should create and use your own application program files. Programs should be installed on the system program file only if specific characteristics, such as memory residency, are required. Using your own program files will increase application transportability, and make revision of the operating system easier.

Since an assembly language program can be constructed in several segments, you must consider the structure of the program when you perform the install operation(s). If the program consists of only a task segment, use the Install Task (IT) command. If the program is a task segment with attached procedures or overlays, you must install the procedures and overlays separately. Use the Install Procedure (IP) command for procedure segments, and the Install Overlay (IO) command for overlay code. If the task is a real-time task, you must use the Install Real-time Task (IRT) command to install it.

If a program contains procedure segments, you must install the procedures first, then the task segment, and then any overlays. Linked object files must contain the separate object modules in that order, so you can install them correctly. Since some of the rules for installing segments of a program also apply to higher-level languages, the subject is discussed in more detail in a separate section.

5.3.5 Executing Assembly Language Programs

There are basically three ways to execute assembly language programs, as follows:

- Bid them from a batch stream using the .BID, .QBID, .DBID or .TBID primitives
- Bid them interactively using the XT, XTS, or XHT SCI commands
- Bid them from another task using an SVC

Since any activation of a program can be referred to as a task, the following discussion uses the term task rather than program. In this context, executing a task is not limited to activation of code installed as a task segment only. The task can have attached procedures and overlays.

The .BID, .QBID, .DBID, and .TBID commands are SCI primitives. Primitives are the lowest-level components of the SCI language as described in Section 6. Tasks initiated by the .BID task execute in foreground and can pass synonyms from the terminal communications area (TCA) to other tasks. The .QBID primitive bids a task in background. The .DBID primitive bids a task in background mode in an unconditionally suspended state, and waits to be activated by a debug command. The .TBID primitive bids a task and terminates SCI so that the SCI task does not occupy system resources. SCI must be activated when the task terminates. Refer to Section 6 for further information.

The XT, XTS, and XHT SCI commands are issued either interactively from a terminal or from a batch stream. The XT command executes a task in deep background. In deep background, a task is not associated with SCI in either background or foreground. SCI remains active at the terminal. Since this allows you to continue working at the terminal, use XT only to execute tasks that require no terminal affiliation or terminal I/O. Send I/O to files or special devices.

The XTS command executes a task and suspends SCI at the terminal until the task terminates. Use XTS to execute tasks that interact with the terminal to avoid contention between SCI and the task for terminal access. The XHT command executes and halts the task immediately, by placing the task in memory in a suspended state. This is useful for debugging, since you can assign breakpoints before reactivating the task. Each of the commands you use for executing tasks are detailed in Volume II.

5.3.6 Assembly Language Programming Exercise

The brief assembly language program given in this section displays a message and requests the input of three numbers. The procedure given here assumes that a VDT is used. It is also assumed that a printer (LP01) is available.

1. Enter the program into the computer.

    a. Assuming you have powered up and logged on, invoke the Text Editor by executing the XE command.

    b. Since you have no previous source file to edit, press the Skip key to bypass the FILE ACCESS NAME parameter. If you were reentering the source file to correct compilation errors, you would specify the pathname where you stored the source code.

c. Enter the following sample program:

```
************************************************************
*                   DATA SECTION                           *
************************************************************
       IDT 'RESPONSE'
************************************************************
*    OPENING DATA WORDS                                    *
*      1. WORKSPACE POINTERS                               *
*      2. PC VALUE AT START OF PROGRAM                     *
*      3. END ACTION ITEM                                  *
************************************************************
       DATA WSP          WORKSPACE POINTER ADDRESS
       DATA START        PC AT PROGRAM BEGINNING
       DATA 0            END ACTION (NONE SPECIFIED)
WSP    BSS  32           WORKSPACE REGISTERS
OPEN   DATA 0            I/O REQUEST
       BYTE >0,>20       OPEN-REWIND LUNO >20
       DATA 0
       DATA 0
       DATA 0
       DATA 0
MSSG0  DATA 0            I/O REQUEST
       BYTE >B,>20       WRITE ASCII ON LUNO >20
       DATA 0
       DATA GREET        MESSAGE LOCATION
       DATA 0
       DATA MSSG1-GREET  MESSAGE LENGTH
************************************************************
* SPECIFY THE FIRST MESSAGE                                *
************************************************************

GREET  DATA >0A0D
       TEXT 'HELLO, PLEASE INPUT NUMBER OF ITEMS'
       TEXT ' SOLD TODAY.  USE 4-DIGIT NUMBERS.'
       DATA >0A0D
MSSG1  DATA 0            I/O REQUEST
       BYTE >B,>20       WRITE TO LUNO >20
       BYTE 0,>40        RESPONSE FOLLOWS MESSAGE
       DATA ITEM1
       DATA 0            CHARACTERS SPECIFIED IN INPUT ROUTE
       DATA 10           MESSAGE LENGTH
       DATA STR1         LOCATION OF INPUT PARAMETERS
STR1   DATA STORE        SAVE PARAMETERS IN STORE
       DATA 4            STORE 4 CHARACTERS
       DATA 0            CHARACTER COUNT AFTER INPUT
STORE  BSS  12
ITEM1  DATA >0A0D
       TEXT 'ITEM 1'
MSSG2  DATA 0            I/O REQUEST
       BYTE >B,>20       WRITE TO LUNO >20
       BYTE 0,>40        READ AFTER WRITE
       DATA ITEM2
       DATA 0
       DATA 10           MESSAGE LENGTH
       DATA STR2
STR2   DATA STORE+4      2ND ITEM CHARACTERS STORE LOCATION
       DATA 4            STORE FOUR DIGITS
       DATA 0
ITEM2  DATA >0A0D
       TEXT 'ITEM 2'
MSSG3  DATA 0            I/O REQUEST
       BYTE >B,>20       WRITE TO LUNO >20
       BYTE 0,>40        READ AFTER WRITE
       DATA ITEM3
       DATA 0
       DATA 10
       DATA STR3
```

```
STR3    DATA STORE+8        3RD ITEM STORE LOCATION
        DATA 4
        DATA 0
ITEM3   DATA >0A0D
        TEXT 'ITEM 3'
MSSG4   DATA 0              I/O REQUEST
        BYTE >B,>20         WRITE TO LUNO >20
        DATA 0
        DATA GOODBY         MESSAGE LOCATION
        DATA 0
        DATA CLOSE-GOODBY MESSAGE LENGTH
************************************************************
* FINAL MESSAGE                                            *
************************************************************
GOODBY DATA >0A0D
        TEXT 'THANK YOU.  THAT COMPLETES TODAY'
        BYTE >27
        TEXT 'S TRANSACTIONS.'
        DATA >0A0D
CLOSE   DATA 0              I/O REQUEST
        BYTE 1,>20          CLOSE LUNO >20
        DATA 0
        DATA 0
        DATA 0
        DATA 0
EOP     BYTE >16,0          TERMINATE TASK
*
START   XOP  @OPEN,15       OPEN LUNO >20
        XOP  @MSSG0,15      OPENING MESSAGE
        XOP  @MSSG1,15      INPUT 1
        XOP  @MSSG2,15      INPUT 2
        XOP  @MSSG3,15      INPUT 3
        XOP  @MSSG4,15      EXIT MESSAGE
        XOP  @CLOSE,15      CLOSE FILE, UNLOAD/REWIND
        XOP  @EOP,15        TERMINATE TASK
        END
```

d. Press the Command key and enter QE, and then press the Return key to quit the Text Editor. Specify NO to the ABORT? prompt, and supply an appropriate pathname for the source code to be stored under.

2. Assemble the program.

   a. Invoke the macro assembler by entering XMA. Supply the pathname where the source is stored, the pathname for the object output and a pathname for the assembler listing. The following shows example pathnames, which place the listings on the system disk.

```
       SOURCE ACCESS NAME:  .SOURCE
       OBJECT ACCESS NAME:  .OBJECT
      LISTING ACCESS NAME:  .LNKOUT
        ERROR ACCESS NAME:  (Press the Return key)
                  OPTIONS:  (Press the Return key)
  MACRO LIBRARY PATHNAME:  (Press the Return key)
              PRINT WIDTH:  80
              PAGE LENGTH:  60
```

b. The assembler runs in background mode. Enter a WAIT command to wait for completion of the assembly. When the assembly completes, the following message appears:

   MACRO ASSEMBLY COMPLETE, nnnn ERRORS, nnnn WARNINGS

   Press the Return key to reenter the command mode.

3. Link the object code.

   a. First create a command file for the Link Editor. Invoke the Text Editor by entering XE. Press the Skip key in response to the FILE ACCESS NAME prompt, then press the Return key twice.

   b. Enter the following Link Editor control file:

```
TASK LANGTST
INCLUDE .OBJECT
END
```

   c. Press the Command key, enter QE, and then press the Return key to quit the Text Editor. Respond NO to the ABORT? prompt, and supply a pathname where the Text Editor can store the control file. You can respond to the rest of the prompts with the Return key.

   d. When the SCI prompt returns, invoke the Link Editor by entering XLE and pressing the Return key. Supply the pathname of the control file you created and the pathname of a file where the results of the linking process can be placed by the Link Editor. Specify another pathname for the listing of the linking process. Accept the print width and page length defaults by pressing the Return key.

   e. The Link Editor executes in background mode, so you can enter WAIT and press the Return key to wait for the linking process to complete. When the Link Editor terminates, the following message appears:

   LINK EDITOR COMPLETED, 0 ERRORS, 0 WARNINGS

   f. Press the Command key to return to SCI.

4. Install the program.

   a. Enter the Install Task (IT) command to place the program on the system program file. (Zero causes the IT command to use .S$PROGA as the program file. We are using this file for simplicity, but do not necessarily recommend using the system program file for user programs.) Specify the following parameters:

```
    PROGRAM FILE OR LUNO:  0
               TASK NAME:  LANGTST
                 TASK ID:  0
OBJECT PATHNAME OR LUNO:   .OBJECT
                PRIORITY:  4
     DEFAULT TASK FLAGS?:  YES
    ATTACHED PROCEDURES:   NO
```

   The system provides the installed ID, and displays it in the following form when the installation completes:

```
TASK NAME = LANGTST
TASK ID = nn
```

   Press the Command key to return to SCI.

5. Execute the program.

   a. Since the program uses LUNO > 20, that LUNO must be assigned to the VDT. Use the Assign LUNO (AL) command and respond as follows:

```
          LUNO:  >20
   ACCESS NAME:  ME
 PROGRAM FILE?:  NO
```

   The message ASSIGNED LUNO = >20 appears. Press the Command key to return to SCI.

   b. Execute the program using the Execute Task and Suspend (XTS) SCI command. Select the following parameters:

```
PROGRAM FILE OR LUNO:  0
     TASK NAME OR ID:  LANGTST
               PARM1:  0
               PARM2:  0
          STATION ID:  ME
```

The test program now executes and displays the following on the screen:

```
HELLO, PLEASE INPUT NUMBER OF ITEMS SOLD TODAY. USE 4-DIGIT NUMBERS.

ITEM 1
```

Enter a four-digit number. The following is then displayed:

ITEM 2

Enter a four-digit number. The following is then displayed:

ITEM 3

Enter a four-digit number. The following is then displayed:

THANK YOU. THAT COMPLETES TODAY'S TRANSACTIONS.

The message is displayed briefly, then control returns to SCI. Delete the task entry by using the Delete Task (DT) command as follows:

```
DELETE TASK
        PROGRAM FILE OR LUNO:  0
               TASK NAME OR ID:  LANGTST
```

Use the Delete File (DF) command to delete the following:

```
.SOURCE
.OBJECT
.LNKOUT
.CNTRLINK
```

6

SCI Programming Language

6.1 INTRODUCTION

The SCI programming language is available for creating SCI commands to support applications, or for modifying existing commands to better support your particular operations. If you are not familiar with SCI and SCI commands, refer to Volume II before continuing with this section.

All SCI commands shipped with the DX10 system are written in the SCI programming language. Programming in SCI is logically similar to programming with common high-level languages. The SCI programming language is composed of commands called *primitives*, variables called *synonyms*, *field prompts*, and *keywords*, and a syntax structure. Together, these components provide a convenient way of tailoring menus and SCI commands to meet your application needs.

The following paragraphs provide background information that is helpful in understanding a discussion of the SCI programming language. This section discusses how to create and modify SCI command procedures, processors, and menus, and then presents a complete guide to the SCI programming language itself. The final portion of the section describes the environment in which the SCI command procedures and processors operate, including procedure file and directory names, batch stream operation, and common error messages.

6.1.1 What an SCI Command Procedure Is

When you enter a valid SCI command into the system in response to the SCI prompt ([ ]), you initiate the execution of an SCI command procedure. The command procedure is similar to a program since it can request information from you interactively, then take the information and perform a single operation or several specific operations.

An SCI command procedure can execute other SCI commands just as a program can execute subroutines. It can also place tasks in execution through the .BID, .QBID, .DBID, or .TBID primitives, and invoke menus to be displayed on the terminal screen. These menus and the SCI command procedures, which are composed of the commands and syntax of the SCI programming language, are described later in this section.

To create a new SCI command, you must create an SCI command procedure, and install it in a procedure library or directory. Create a sequential file containing valid statements composed of the SCI language, and place that file in a standard directory that you have designated to contain only command procedures and menus.

A command procedure collects the necessary input by displaying field prompts on the screen and receiving operator input, or by recognizing input coded in the procedure itself. It then sets up the environment required for the operation, and finally bids a pre-written task to complete the operation. The task, called a command processor, can be a user-written task, or one of the system tasks. The command procedure uses one of the bid primitives (.BID, .QBID, .DBID, or .TBID) to place the task into execution. The parameters needed by the task are passed to it from the command procedure.

The following example performs a list directory function. It requests the pathname and listing access name from the user, and then bids a task with ID >32 to do the listing operation.

```
LD(LIST DIRECTORY),
PATHNAME=ACNM,
LISTING ACCESS NAME=*ACNM
.BID   TASK=>32,CODE=>32,
PARMS=(>32,@&PATHNAME,@&LISTING ACCESS NAME)
```

The first line of the example defines the name of the command procedure. The next two lines display prompts on the screen by specifying a character string to be displayed, and equating it with the ACNM prompt designation. The ACNM specification requires that the input be a valid pathname or device. The next line bids task >32 and passes the parameters required for the listing operation through the CODE and PARMS keywords. The parameters are supplied by the operator in response to the field prompts. The ampersands (&) in the PARMS list denote field prompt names.

6.1.2 What a Procedure Directory Is

A procedure directory is any directory containing SCI command procedures. For example, the directory .S$PROC on the system disk contains all the system SCI command procedures. (You can view the .S$PROC directory using the LD command.)

When an SCI command is executed, SCI searches the .S$PROC directory for a file name matching the command name. If it exists, SCI executes the file; if not, an error is returned.

When you create several related SCI commands, store them in a separate command procedure directory. Use the .USE primitive to specify this new directory as one of the user directories. SCI then automatically searches this new directory as explained later. Refer to the discussion of the .USE primitive for more information.

To create a command procedure directory, use the standard Create Directory File (CFDIR) SCI command. Further details are given in subsequent paragraphs.

6.1.3 What an SCI Command Processor Is

An SCI command processor is any task placed into execution from an SCI command procedure. The task can can be a simple routine or an entire applications program, and can be written in assembly or any high-level language supported by DX10. You can use the .BID .QBID, .DBID or .TBID primitive in a command procedure to place a command processor task into execution.

SCI processors can access system processor subroutines that perform particular operations, or provide the interface between procedures and processors. For example, the S$STOP processor subroutine terminates the processor, and allows SCI to resume in an orderly fashion. Always include this subroutine at the end of any program used as a command processor. The available system processor subroutines are described in the next paragraph.

6.1.4 What a Processor Subroutine Is

There are three types of processor subroutines included with DX10, and available for use by an SCI processor. These types are:

- String utility subroutines
- SCI interface subroutines
- Terminal local file display subroutines

String utility subroutines operate on character strings passed from SCI to the particular routine. You can use these subroutines to convert binary integers to ASCII, convert from ASCII to binary integer, compare two strings, or copy a string into another address.

SCI interface subroutines allow a command processor to access its parameters from the command processor (that is, its PARMS and CODE values), to locate and modify the synonyms defined for the terminal, and to return control to SCI. The parameters they access are located in the terminal communications area (TCA), so these subroutines include an S$GTCA operation (Get TCA), and an S$RTCA operation (Release TCA). The remaining interface subroutines can be issued any time after an S$GTCA call and before an S$RTCA call.

Terminal local file display subroutines open, close and construct records in the terminal local file (TLF). Special routines are required because the TLF is a file of ASCII data and is accessed by SCI itself. Whenever you wish to place data or text on the terminal screen, you must open the TLF, write records to it and close it when you finish.

The available processor subroutines are discussed individually later in this section.

6.1.5 What the SCI Language Is

The SCI language is a programming language made up of commands and data definition statements. These statements direct the operation of DX10 according to the structure of the command that they compose. The components of the SCI language are as follows:

- Primitives
- Variables
- Special characters

Primitives are the basic unit of the SCI language. They are predefined operators recognized by SCI as verbs, and are similar to verbs in high-level languages. These primitives perform such diverse functions as logical operations, data manipulation, control, and task activation. The format of a primitive is a verb, preceded by a period (.). Examples of primitives are .IF, .SHOW, .STOP, and .BID. SCI primitives are discussed in more detail later in this section.

Variables in the SCI language are similar to variables in high-level languages. You can assign different values to them according to the requirements of the command procedure. The three types of variables in the SCI language are field prompts, synonyms, and keywords. Variables are explained in detail later in this section.

Certain special characters are recognized by the SCI language as indicators and delimiters. For example, an asterisk (*) in column one indicates that the line is a comment line; if it is in any other column, it denotes an optional field prompt. An exclamation mark (!) marks the end of a record. All valid special characters are defined later in this section.

6.2 IMPLEMENTING COMMAND PROCEDURES, PROCESSORS, AND MENUS

The following paragraphs provide a further overview of the SCI programming language by explaining how to create and implement new command procedures, processors, and menus. The steps required to complete these processes allow a better understanding of the details of the programming language itself.

6.2.1 Creating and Changing SCI Command Procedures

To create a working SCI command procedure, you must write the command procedure using the SCI language and syntax, and then install the procedure in a procedure directory. The procedure directory must already exist. If you want to create your own procedure directory, do so before creating command procedures. There are three ways to create a command procedure:

- Using the Text Editor
- Using the .PROC primitive interactively
- Using a batch stream

These are explained in the following paragraphs.

6.2.1.1 Using the Text Editor. Use the Text Editor to enter the written procedure into the system. Install the procedure by storing it under a procedure directory when you quit the Text Editor. The file name you store it under must be identical to the procedure name specified on the first line of the procedure code. That name can be up to eight characters long, and can contain the special character $.

Figure 6-1 is an example of a command procedure that performs a show file function. The procedure name, SF, must be defined on the first line. If you are using the Text Editor, you must specify a pathname such as VOL1.SAMPLE.SF when you quit the Text Editor, to indicate where the system stores the new file. In the example pathname, SAMPLE represents the name of the command directory. The file name, SF, is identical to the name defined on the first line of the command procedure itself.

```
SF (SHOW FILE),
    FILE PATHNAME = ACNM(@$SF$P)
    .SHOW &FILE PATHNAME
```

Figure 6-1. SF Command Procedure Example

When you execute this command procedure, by entering SF in response to the SCI prompt ([ ]), the first line of the command procedure displays along with the prompts. This display serves to identify the command procedure to the user.

6.2.1.2 Creating Command Procedures Interactively. You can also create new command procedures interactively. Use the .PROC primitive to alert SCI that you are entering lines in a command procedure, as shown in the following example. The first step in this example creates a new procedure directory, which may not be necessary in every case. Once you have entered the .PROC primitive, SCI accepts strings until it receives the .EOP primitive. Note that this example is not intended to create an executable procedure.

```
[ ]CFDIR  PATHNAME=SYSVOL.MYLIB,MAX=101
[ ].USE   SYSVOL.MYLIB,.S$PROC
[ ].PROC  EXP(EXAMPLE PROCEDURE),
          EXAMPLE NAME=NAME,
          NUMBER=INT
  .BID    TASK=>55,PARMS=("&EXAMPLE NAME","&NUMBER"),
  .EOP
[ ]next SCI command
```

A new command procedure name must be unique within the procedure directory where it resides. You can create a command procedure with the same name as an existing SCI procedure only if you place it in a separate procedure directory. If you modify an existing command procedure to perform alternate functions, do not store the new command procedure in the .S$PROC directory, unless you plan for it to completely replace the standard SCI command by that name. Complete replacement is not recommended in most cases.

The .EOP primitive signals the end of that command procedure, and SCI begins recognizing input as SCI commands. You can then issue any valid SCI command, including another .USE or .PROC primitive to begin another command procedure.

If you have many command procedures to enter into the system, this method becomes rather time consuming.

6.2.1.3 Using a Batch Stream. Batch streams are files that contain standard SCI commands. The system reads these commands from the file just as if they were issued by a user at a terminal. The prompt responses are coded into the batch stream itself. To create a command procedure using a batch stream, you create a batch stream file that contains the commands described previously for creating them interactively. This is very useful if you want to install several new procedures in different procedure directories.

Batch streams and batch stream operation are explained in detail in a later paragraph.

6.2.1.4 Creating a Procedure Directory. A procedure directory can be any directory of sequential files. The system's procedure directory is .S$PROC. All SCI commands that are shipped with DX10 are sequential files in the .S$PROC directory. Although you can add procedures to the .S$PROC directory, it is usually more convenient to create your own procedure directories, according to the requirements of your applications. This practice can also make transporting applications easier.

To create user procedure directories, use the CFDIR SCI command. Specify a maximum size approximately 10% larger than the anticipated size to minimize access time. If you want the directory on the system disk, specify a pathname beginning with a period (.), since pathnames beginning with a period always reference the system disk. If you want the directory on a separate volume, indicate the volume name as the first item in the pathname. If your applications are modular, it is usually most convenient to have the procedure directory accompanying the application programs that use the commands.

Sometimes a procedure directory is called a procedure library, since it contains a collection of functionally similar files. References to procedure libraries usually occur in error messages relating to directory/procedure discrepancies. For example, if you store a new procedure under a file name different from the name you gave it on the first line of the procedure, the following message appears when you attempt to execute it:

    **** ERROR FF02 **** PROCEDURE LIBRARY ERROR

The .USE primitive ensures that you always access the correct procedure directory. The .USE primitive can have from one to five procedure directories as parameters. Within a command procedure, you can issue a .USE primitive before bidding another procedure. SCI searches the directories in the order specified. You can also specify a different procedure directory for interactive SCI input by executing the .USE primitive in response to the SCI prompt ([ ]) and specifying the desired directories, or by modifying the log-on command procedure as discussed in the next paragraph. The .USE primitive can be invoked interactively or within a command procedure such as the .S$PROC.M$00 log-on procedure.

6.2.1.5 Naming New Command Procedures. A new command procedure name must be unique within the procedure directory where it resides. You can create a command procedure with the same name as an existing SCI procedure only if you place it in a separate procedure directory. If you modify an existing command procedure to perform alternate functions, do not store the new command procedure in the .S$PROC directory, unless you plan for it to completely replace the standard SCI command by that name. Complete replacement is not recommended in most cases.

Before creating a new command procedure, review the .S$PROC procedure directory using the LD SCI command. If the name you plan to use does not already exist, you can place it in the system procedure directory, or an alternate directory.

To modify existing command procedures, change the command procedure file. Invoke the Text Editor, and specify the pathname of the command procedure you want to change. Modify it as required, and save it back in the same file. To move it from one procedure directory to another, use the Copy Concatenate (CC) SCI command to copy the command procedure file to the new, existing directory, and then delete the file from the old directory.

6.2.2 Creating and Changing SCI Command Processors

When creating new command processors, you should identify all the activities the processor should perform. It may be more desirable to use several command processors if the function is long, or requires several separate operations. Each processor can then be called from the same command procedure.

You should also determine whether you want the processor to execute in background or in foreground. Processors that require interaction with the user, or produce output for the terminal should execute in foreground, and be placed into execution by a .BID primitive in the command procedure. If no terminal interaction is necessary, you can use the .QBID primitive, which executes a task in background. These primitives are explained later in this section.

Before a command processor can execute, you must link it with the interface subroutines and install it on a program file. To link a command processor with the interface subroutines you must specify SCI990.S$OBJECT as a library when you link the processor. This library contains all the interface subroutines. Also, command processors can access the synonym table. Access to both PARMS and synonyms is through the TCA. The task can return synonym values and completion codes or messages. The task itself does not need to be in assembly language, but access to the required S$ routines must be gained either through high-level language runtime routines or a user-written assembly language interface routine.

6.2.3 Creating and Changing SCI Command Menus

SCI menus are special-purpose files referenced by command procedures. They contain a formatted display to appear on the terminal screen. You create or change the display using the Text Editor.

You activate a menu interactively by a single word, preceded by a slash (/). SCI uses the slash to determine if the following string is a menu name. The .MENU primitive, described later, displays the menu file on the screen.

Menu file names must begin with the M$ prefix, and must reside in the same command procedure directory as the command procedures they support. SCI recognizes the M$ prefix as a menu file, and processes it appropriately.

An example of a menu is the main SCI menu that appears after you log on to a terminal. When you select one of the available command groups, a sub-menu appears. When you enter a specific command, that command executes. If you are unfamiliar with the menu concept in DX10, refer to the discussion of menus in Volume I before continuing here.

To create menus to support your applications, create a file and place up to 23 lines of data in it. That data will appear just as you enter it when that menu is activated.

The SCI command menus reside in the .S$PROC command directory. The main menu is named M$LC, and the submenus are generally named according to the command group they support. For example, the submenu displayed when you specify the /EDIT command group from the main menu is named M$EDIT. If you want a new menu to be displayed at log-on, rather than the SCI top-level menu that is currently the default, you can modify the log-on command procedure as described in the next paragraph.

6.2.4 Log-On/Log-Off Command Procedures

When you log on, SCI searches for a command procedure called M$00 residing under the pathname .S$PROC.M$00 (on the system disk). This procedure is not shipped on the DX10 operating system disk, but you can create it and use it to include special log-on processing.

For example, if you want to change the default menu from the main SCI menu, M$LC, to another menu, you can create an M$00 command procedure and change the .USE option to select a directory containing another menu named M$LC. You can also change the default procedure directory by modifying M$00. You can specify that another directory be searched before or after the .S$PROC directory, or you can specify two user procedures to be searched instead of .S$PROC.

When you log off the system, SCI executes the M$01 command procedure. If you want to change or add to the log off activities performed by SCI, modify the M$01 command procedure.

6.2.5 SCI .S$NEWS File

After completing the initial log-on procedure, SCI searches for a file on the system disk named .S$NEWS. If it finds the .S$NEWS file, it displays it on the terminal screen before displaying the default menu.

Since the entire file is displayed as it exists, you can use it to convey messages to all terminal users. .S$NEWS is a sequential file, so you can place messages in it using the Text Editor. You can use .S$NEWS instead of a message to the terminal, because you can be sure that each user who logs on has the opportunity to see it.

6.3 SCI LANGUAGE

The SCI language functions similarly to any other high-level language. It is composed of primitives, variables and special characters, that, when used in the correct command format, function together to make a logical programming language.

The following paragraphs describe the SCI language. Command format and special characters are reviewed first, since they are rather simple concepts. Then variables are discussed in detail, and each primitive available in the SCI language is presented. Finally, each SCI command processor subroutine is described. Several procedure examples are included to help you understand how the elements of the language work together.

6.3.1 Command Format

The command format described here is required for statements in the SCI programming language. SCI commands entered interactively in response to the SCI prompt ([ ]) always take the form of 1-8 alphabetic characters and possibly an imbedded dollar sign, and may additionally employ an elongated form called "expert mode" described in Volume II.

Commands within an SCI command procedure must conform to certain format constraints. There are two basic command formats in the SCI language, command format and primitive format.

Command format is used within a procedure to assign values to character strings that are prompts. It assumes the following general form:

<command> <blank(s)> <field prompt assignments>

An example of the command format is as follows:

```
IDT       YEAR=1982,MONTH=4,DAY=27,HOUR=18,MINUTE= 56
```

This example duplicates the effect produced if you typed in the IDT command in response to the SCI prompt ([ ]), and then responded to the prompts displayed. Since there are several types of prompts in the SCI language, they are discussed later in more detail.

Primitive format is used for all other lines in a command procedure. It assumes one of the following general forms:

<primitive> <blank(s)> <keyword list>

<primitive> <blank(s)> <single argument>

Examples of the primitive format are as follows:

```
.PROC    NM(NEW MENU PROC)

.STOP    TEXT = NORMAL TASK COMPLETION,
         CODE = 3

.SYN     MY = VOL2.SMITH.SOURCE

.SHOW    @MY.DATA1
```

In the first example, the primitive .PROC specifies a new procedure is being created. The .PROC primitive does not require keywords. Instead, it requires a single argument indicating the name of the new procedure. The argument must be in a special format as required by the .PROC primitive.

In the second example, the primitive .STOP indicates the end of execution in a command processor. The keyword TEXT contains the termination message to be returned to the procedure, and the keyword CODE returns a user-defined termination code.

In the third example, the keyword MY is designated as a synonym by the .SYN primitive. The value of the synonym keyword is a pathname.

In the fourth example, the primitive .SHOW is issued. This primitive has associated keywords, but needs a single argument to indicate the name of the file to be shown. The argument in this example uses the synonym assigned in the third example. The special character @ indicates a synonym that needs to be resolved. In this example, MY is resolved to the pathname VOL2.SMITH.SOURCE. (Table 6-1 details all valid special characters and their meanings.)

A command format can span several lines through the use of continuation characters such as the comma. However, there cannot be more than one command format on one line.

All valid primitives are explained in detail later in this section. The special characters used in the SCI programming language are discussed in the following paragraphs.

6.3.2 Special Characters

The commands and syntax of the SCI programming language are greatly expanded through the use of a small set of special characters. They are used in conjunction with other valid elements of the language, such as commands, field prompts, primitives, and keyword lists described above to create powerful SCI command procedures.

Table 6-1 lists the valid special characters and their meanings.

Table 6-1. SCI Special Characters

Character	Usage
!	Indicates end of record. Comments may occur after an ! character.
*	If in column 1, indicates a comment statement. If preceding a valid prompt type, indicates that field prompt response is optional.
@	Indicates that the character string following the @ is a synonym. If the synonym was previously defined, this causes the @ sign and character string to be replaced by the synonym value. Otherwise, the value of the synonym is defined as the character string.
&	Indicates that the character string following the & is a field prompt name, to distinguish it from synonyms and literals. SCI replaces the & and the character string with the field prompt value. If no value is assigned, a null string results.
Λ	Delimits synonyms.
=	If in column 1, causes the line in a batch stream to execute, but not written to the listing file. Also used to assign values to keywords or field prompt names. A language line ending with = is automatically continued.
,	Delimits elements of a keyword list or field prompt assignments. If it appears as the last character on a command line, indicates continuation of that line.
( )	When placed around a field prompt type, indicates that the response can be a single item or a list. (Within a batch stream, the response list must also be enclosed in parentheses). When placed around a character string that follows a field prompt name, indicates that the string appears as an initial value for the prompt.
" "	Used with character strings representing a synonym, or that otherwise would be ambiguous. Use them to enclose strings containing the @ sign, or other special characters to be resolved literally. When used with the @ sign, they indicate that the synonym resolves to a list of items.
.	Indicates the next character string is an SCI primitive.

Examples using these special characters are contained throughout the discussion of the SCI programming language, and at the end of the section. When studying the examples, refer often to Table 6-1. This way you can see the effect the special character has when used with the particular primitive or variable (field prompt, keywords, synonyms, and so on) being presented at the time.

6.3.3 Variable Types

The three types of variables in the SCI programming language are as follows:

- Field Prompts
- Synonyms
- Keywords

The following paragraphs discuss these variable types.

6.3.4 Field Prompts

Field prompts are used in command procedures that interact with the terminal user. They function as variables, and the user assigns the values. The field prompts you specify in a command procedure are the same ones that are displayed on the screen when the command procedure executes.

When you execute an SCI command, that command requires you to supply additional information, based on the function of the command. This information can be in several forms. For example, you can enter pathnames, integers, or strings like "YES" or "NO." Sometimes if you enter the wrong type of data in response to a prompt, an error results, because the data you enter must be compatible with the characteristics of the field prompt type specified in the command procedure. Valid SCI field prompt types are shown in Table 6-2.

Table 6-2. Valid Field Prompt Types

Prompt Type	Format	Value Restrictions
ACNM	ACNM ACNM(initial value)	File pathname or device name; include optional initial value
INT	INT INT(initial value)	Integer value; include optional initial value
NAME	NAME NAME(initial value)	Letters, numbers, or $ sign; include optional initial value
YESNO	YESNO YESNO(initial value)	Response beginning with Y or N; include optional initial value
STRING	STRING STRING(initial value)	Alphanumeric or special characters; include optional initial value

The following list of rules governs field prompt types:

- An asterisk (*) preceding the field prompt type makes the response optional. The user can press the Return key without entering a response.
- Parentheses around a string that follows the field prompt specification makes that string appear on the screen as an initial value for that prompt.
- Initial values that resolve to a character string beginning with the dollar sign ($) resolve to a null string.
- The response to a field prompt can be a single value or a list of values. To indicate that the response can be a list, enclose the prompt type in parentheses. For example, INPUT=(ACNM) indicates that the field prompt INPUT can assume a single value, or a list of values of the ACNM type. Responses to the INPUT prompt can then be either a single item or a list of items. When the you enter a list of items as a response on the screen, separate each item with a comma. When supplying the response list in a batch stream, separate items with a comma, and enclose the entire response list in parentheses.
- Initial values can be specified for all field prompt types. Initial values that are lists must be enclosed in quotation marks.
- Field prompt names can be up to 28 characters in length, including spaces.
- Field prompt names cannot include the following special characters:

  @ At sign

  ^ Caret

  = Equal sign

  & Ampersand

  " Double quote

  ' Single quote

  , Comma

- The following characters affect the operation of the .EVAL primitive and should be avoided in prompt names:

  + Plus sign

  - Minus sign

  / Slash

  * Asterisk

- Prompt responses can be up to 50 characters in length.
- Prompt names referenced within a batch stream or in *expert mode* can be in an abbreviated form. The system resolves these references using a *near equality* algorithm. The near equality algorithm is discussed in more detail following the discussion of field prompt types.

The system's SCI command procedures provide many good examples of how these field prompt types function. You can use the SF command to view any of the SCI command procedures. Some good examples to study are:

- S$PROC.CFDIR
- S$PROC.AS
- .S$PROC. INV
- S$PROC.DF

Each field prompt type is described in the following paragraphs. Additional examples are provided at the end of this section.

6.3.4.1 ACNM Field Prompt Type. The ACNM field prompt type restricts input to a file name or a device name. An example of the ACNM field prompt type is:

FILE PATHNAME = *ACNM("@SFP")

This syntax and special character usage in this example indicates the following:

- The response is optional, since an asterisk appears in front of the field prompt type.
- The response must be a single value, since there are no parentheses around the field prompt type.
- The characters SFP represent a synonym since they are preceded by an at (@) sign.
- The quotation marks around the characters @SFP indicate that the value of the synonym is a list.
- The parentheses around the characters "@SFP" indicate that the value appears as an initial value.

6.3.4.2 INT Field Prompt Type. The INT field prompt type restricts input to an integer value. The expression is a 32-bit hexadecimal integer expression in the range of >80000000 through >7FFFFFFF or decimal integer expression in the range of -2147483648 through 2147483647. The > sign or a leading zero denotes a hexadecimal value. The following is an example of the INT field prompt type:

PARM1 = INT(256)

In this example, the value 256 appears as an initial decimal value.

6.3.4.3 NAME Field Prompt Type. The NAME field prompt type restricts input to alphabetic characters, digits 0-9, and the dollar sign. This input can begin with a dollar sign or alphabetic character. Characters entered in lower case are optionally mapped to upper case by SCI, if the lowercase option was previously selected with the .OPTION primitive. The following is an example of the NAME field prompt type:

```
TASK NAME = *NAME
```

In this example, the asterisk preceding the field prompt type indicates that the user response is optional.

6.3.4.4 STRING Field Prompt Type. The STRING field prompt type restricts input to a string of characters which does not include quotation marks, exclamation marks, equal signs, parentheses, or commas. STRING can also accept a string of characters enclosed by double quotation marks, known as a quoted string. A quoted string can include quotation marks, exclamation marks, equal signs, parentheses and commas. Lower case characters within a quoted string are not mapped to upper case. The following is an example of the STRING field prompt type:

```
INPUT = *(STRING)("@ABCS")
```

The syntax and special characters of this example indicate the following:

- The user resonse is optional, indicated by the asterisk preceding the field prompt type.
- The user response to the field prompt can be a single value or a value list, indicated by the parentheses around the field prompt type.
- The characters ABCS represent a synonym, indicated by the at (@) sign preceding the characters.
- The value of the synonym can be a list of values, indicated by the quotation marks around the characters @ABCS.
- The synonym list is an initial value, indicated by parentheses around the characters "@ABCS".

If you want double quotation marks to appear in the string text, use two sets of double quotation marks in the field prompt assignment line, as follows:

```
INPUT = STRING("PRESS ""RETURN"" TO CONTINUE")
```

In this example, the initial value of the INPUT field prompt reads:

```
PRESS "RETURN" TO CONTINUE
```

6.3.4.5 YESNO Field Prompt Type. The YESNO field prompt type restricts input to an alphabetic character string beginning with either Y or N. The following is an example of the YESNO field prompt type:

ARE YOU SURE? = YESNO

Any response beginning with Y or N is valid, since SCI only examines the first character of the response. The value is determined to be YES if the response begins with Y, and NO if the response begins with N.

6.3.4.6 Abbreviating Field Prompts. SCI allows you to abbreviate field prompt names when referencing them within a command procedure. This is especially useful when building batch streams. SCI uses a near equality algorithm to match abbreviated field prompt references to the correct field prompt. The following rules apply to field prompt abbreviations.

- The field prompt name and the field prompt abbreviation must have the same first character.
- All characters in the abbreviated field prompt must appear in the full field prompt, and appear in the same order.
- Numeric characters in the full field prompt name must also be in the abbreviated field prompt name. For instance, if the full field prompt name is SOURCE1 ACCESS NAME, the abbreviated field prompt must be at least S1. No characters beyond a special character will match if the special character is not in the abbreviated field prompt.
- The first character in the abbreviated prompt that is not in the current word in the full field prompt name must be the first character of the next word in the full field prompt name. The field prompt SOURCE ACCESS NAME could be abbreviated as SOA (the first two characters of SOURCE and the first character of ACCESS), but not as SON.
- Each abbreviation must be unique; it cannot be a valid abbreviation for any other prompt in the procedure. DELETE THE FILE? and DATE TO PRINT can both be abbreviated as DT, but SCI cannot properly identify the intended prompt. In this case, DTTF and DTP would be more appropriate abbreviations.

6.3.4.7 The Dollar Sign in Initial Values. If the initial value specified for a field prompt begins with a dollar sign ($), then that initial value becomes a null string. The following example illustrates this:

INPUT FILE = ACNM(@$ABC)

The prompt has an initial value, which is the value of the synonym $ABC. Ordinarily, if a synonym is not previously defined, the initial value resolves to the synonym name itself, in this case, $ABC. However, since the synonym name begins with a dollar sign, the system resolves the initial value to a null string.

Also, a synonym having an assigned value beginninng with a dollar sign would resolve to a null string. For example:

```
.SYN PATH = $ABC
INPUT FILE = ACNM(@PATH)
```

Since the synonym PATH has an assigned value of $ABC, the initial value of INPUT FILE is resolved as $ABC; then it is further resolved to a null value because of the leading dollar sign.

6.3.5 Synonyms

You are probably already familiar with synonyms and their usage in the interactive SCI environment. Synonyms in the interactive environment provide a means to assign a single word abbreviation to a pathname, and allow you to use that abbreviation instead of entering the entire pathname in response to an SCI command's prompts. Although technically, SCI treats these synonyms the same as the synonyms in the SCI language, they are sometimes used a little differently within a command procedure than in an interactive environment. The accessiblity to synonyms by command procedures gives the procedures great flexibility.

6.3.5.1 Types of Synonyms. There are two kinds of synonyms used in the SCI programming language:

- Synonyms that you both define and assign values to within the command procedure.
- Synonyms that are already defined, and are available for you to use within the command procedure, by assigning your own values.

Synonyms that you both define and assign values to are used similarly to synonyms in interactive SCI. The following example represents possible command lines in a command procedure:

```
.SYN     MY = VOL2.SMITH.SOURCE
.SHOW    @MY.DATA1
.SHOW    @MY.DATA2
```

In this example, the synonym MY is defined and assigned a value of a specific pathname, using the .SYN primitive. Then, the synonym MY is used instead of the full pathname in the .SHOW primitive. It is resolved to the assigned pathname when SCI encounters the @ sign.

Synonyms that are defined by the system are also available to command procedures. These synonyms are used by DX10 to keep track of such things as the last file pathname specified in the Show File (SF) SCI command, or the last line printer you used in a Print File (PF) command. When you execute these SCI commands, and others, the initial value displayed is the last value you used with that command during the terminal session. They are stored as part of the workspace associated with your user ID, and are not deleted until the Q command is executed.

Some of the common system synonyms are given in Table 6-3. You can also review the list of synonyms currently associated with your user ID by executing the List Synonym (LS) command. All system synonyms begin with a dollar sign.

Table 6-3. Standard Synonyms

Synonym	Meaning of Assigned Value
$$MO	A two-digit hexadecimal code for the SCI mode: 00 = Batch mode 01 = TTY mode 0F = VDT mode
$$ST	A two-digit decimal station number
$$UI	A six-character user identification
ME	A four-character station name (for example, ST09)
$$CC	A five-digit hexadecimal code returned by the S$STOP subroutine in foreground
$$BC	A five-digit hexadecimal code returned to the foreground synonym table by the .STOP primitive in a background procedure or by S$STOP from a background task.

An example of the way these synonyms are used is shown by a SF command procedure below:

```
SF(SHOW FILE)
        FILE PATHNAME = *ACNM(@$SF$P)
        .SYN $SF$P = "&FILE PATHNAME"
        .IF "&FILE PATHNAME",NE,""
        .SHOW @&FILE PATHNAME
        .ENDIF
```

6.3.5.2 Synonym Evaluation. When the SF command procedure shown in the preceding example executes, it assigns the synonym SFP a value equal to the character string entered as a response to the prompt FILE PATHNAME. Any procedure that uses the synonym SFP for an initial value automatically uses the file name last specified in the SF command, or whatever procedure or processor last updated the value of the synonym, SFP. In the SF example, the .SYN primitive updates the value of SFP to whatever character string is entered in the prompt FILE PATHNAME.

The reference @&FILE PATHNAME causes SCI to evaluate &FILE PATHNAME as the name of a synonym rather than a data field. For example, your response to the FILE PATHNAME prompt could be MYFILE, which is already defined as a pathname such as VOL1.ME.DATA1. MYFILE gets resolved to the correct pathname. If the pathname is entered instead of a synonym like MYFILE, SCI first attempts to find a synonym with that name, and then searches for a file by that name. This allows you to enter either synonyms or pathnames in response to field prompts.

To delete synonyms in a command procedure, set the value equal to a null value, as follows:

```
.SYN MYFILE = ""
```

When the at sign (@) is encountered, SCI proceeds to resolve the synonym as follows:

- As soon as SCI encounters the @ sign, it tries to match the first portion of the string following the @ with a name in the synonym table. The first portion of the string consists of those characters between the @ sign and another special character other than a dollar sign, and can contain letters, numbers and $ symbols.

- If the string does not have a match, the synonym is currently undefined. SCI assigns it the value of the string itself. In some situations where the synonym is not followed by a special character, you can enclose the synonym with carets (Λ) to facilitate proper substitution. For example, if the synonym USER had a value of ID01 and appeared in the context USERFILE with no special character after USER, you would use the following code to obtain proper synonym substitution:

    @ΛUSERΛFILE

After substitution, the field would appear as follows (with the carets (Λ) removed):

    ID01FILE

You can embed synonyms within pathnames in command procedures.

Responses to prompts may not include special characters such as the caret (Λ) and the at sign (@). Because of this restriction, interactive prompt responses cannot use the full power of synonym evaluation. The preceding example, when entered as a prompt response, returns an error message. Within a procedure, .SHOW @ΛUSERΛFILE is evaluated as .SHOW ID01FILE, according to the rules previously stated.

Because of the significance of special characters in the evaluation of synonyms, the use of synonyms to represent DX10 pathnames can cause problems. If a synonym is used to represent an entire pathname or only the first component of a pathname, no problem exists. Synonyms can be secondary components of a pathname if the @ sign is properly placed in the string evaluation. For example, VOL1.MYCAT.@S is evaluated as VOL1.MYCAT.SOURCE, where S is the synonym for the value of SOURCE.

However, if a synonym is used as a secondary component of a pathname, and the @ sign is placed in front of the pathname, the synonym is not properly resolved. The pathname @VOL1.MYCAT.S is an example.

Procedures that create synonyms for temporary use must delete them to avoid synonym table overflow. This must be done as the last step of any procedure.

6.3.6 Keywords

Keywords are variables used to specify parameters for SCI primitives. Keywords are stored in the keyword table, which occupies the same physical space on disk as the synonym table, along with the value initially assigned in the command procedure. Although keywords are usually required arguments for SCI primitives, the word keyword is sometimes used in error messages to mean prompt.

Not all SCI primitives require keywords. Some primitives require more than one keyword. The .BID primitive requires you to assign values to three keywords, and optionally to a fourth keyword. The keywords for .BID are TASK, and optionally LUNO, CODE, and PARMS. If you do not specify a value for the keyword TASK, the .BID primitive cannot perform the bid function.

The .SPLIT primitive requires you to assign values to two keywords, and optionally, a third. The keywords are LIST, FIRST, and optionally, REST. You must assign values to the first two keywords in order for the primitive to execute properly, while the third keyword allows you additional usage flexibility. These keywords and the values you assign to them become the parameters of the primitive.

A keyword can be assigned several values, known as a list of values. The keyword LIST in the .SPLIT primitive can have a series of elements specified as its values, with each element separated by a comma.

The function of keywords becomes more obvious after you begin studying the SCI primitives.

6.3.7 SCI Primitives

SCI primitives are the basic members of the SCI programming language. They are similar to verbs or operators in high-level languages since each primitive initiates a specific function, and demands a properly formatted argument or keyword list.

Each SCI primitive is discussed in the following paragraphs. Table 6-4 lists the available primitives and their formats. With the exception of the .PROC primitive, the primitive definitions are in alphabetical order with related primitives grouped together. In the following table, items enclosed in square brackets [ ] are optional format components. Items in lowercase and enclosed in angle brackets < > are response types for each argument component.

Table 6-4. SCI Primitives

Primitive Command	Parameters
.PROC	< name> [(< full name>)][= < int>][,< prompt list>]
.EOP	
.BID	TASK = < name/int> [,LUNO = < int>][,CODE = < int>] [,PARMS = (< string,...,string>)]
.QBID	
.DBID	
.TBID	
.DATA	< acnm> [,EXTEND = < YES/NO>][,SUBSTITUTION = < YES/NO>] [,REPLACE = < YES/NO>]
.EOD	
.EVAL	<synonym> = <value>
.EXIT	
.IF	< op1> ,< relation> ,< op2>
.ELSE	
.ENDIF	
.LOOP	
.UNTIL	< op1> ,< relation> ,< op2>
.WHILE	< op1> ,< relation> ,< op2>
.REPEAT	
.MENU	[< menu name>]
.OPTION	[PROMPT = < string>][,MENU = < name>] [,PRIMITIVES = < YES/NO>] [,LOWERCASE = < YES/NO>]
.OVLY	Reserved for system use
.PROMPT	[(< full name>)][= < int>][,< prompt list>]
.SHOW	< acnm> [,< acnm,...>]
.SPLIT	LIST = (< string> ,< string> ...< string>), FIRST = < synonym> [,REST = < synonym>]
.STOP	[TEXT = < string>][,CODE = < int>]
.SVC	Reserved for system use
.SYN	< name> = "< value> "
.USE	[<acnm1>][,<acnm2>][,<acnm3>][,<acnm4>][,<acnm5>]

6.3.7.1 .PROC and .EOP Primitives. The .PROC primitive signals SCI that the character strings following the .PROC primitive are part of a command procedure. Use .PROC to install a command procedure onto a command procedure library through a batch stream or interactively through the terminal. If you are using the Text Editor to create a command procedure, the .PROC primitive is not required.

The .PROC primitive has the following format:

.PROC <name> [(<full name>)][=<int>][,<prompt list>]

The name parameter is required. It defines the name of the procedure, and may be up to eight characters in length. It cannot contain any special characters except the dollar sign. The numbers 0 through 9 and the dollar sign cannot be the first character of a procedure name. An optional full name, enclosed in parentheses, may be given immediately following the procedure name. The full name is displayed on the terminal when the procedure is invoked. The name of the file where you save the command procedure, and the name you specify within the command procedure as the procedure name must be the same.

A new command procedure name must be unique within the procedure directory where it resides. You can create a command procedure with the same name as an existing SCI procedure only if you place it in a separate procedure directory. If you modify an existing command procedure to perform alternate functions, do not store the new command procedure in the .S$PROC directory, unless you plan for it to completely replace the standard SCI command by that name. Complete replacement is not recommended in most cases.

An optional field, [=<int>], can follow the full procedure name to specify the desired privilege level of the command. Table 6-5 shows the valid privilege levels and their meanings.

Table 6-5. Command Privilege Levels

Level	Meaning
0	Lowest level of access privilege. For example, the Show File (SF) command can be accessed by all users.
1	User-defined
2	System access level. For example, the Kill Task (KT) command requires experienced user access only.
3	User-defined
4	Management access level. For example, only system management or system security personnel can access the Assign User ID (AUI) or Modify User ID (MUI) commands.
5	User-defined
6	Combination of System and Management. For example, the Execute System Generation Utility (XGEN) command requires restricted access.
7	User-defined

Command privilege levels prevent commands from being invoked by a user having a privilege level lower than the level assigned to a particular command. (The user's privilege level is assigned using the Assign User ID (AUI) or Modify User ID (MUI) SCI commands.) The privilege level allows you to limit access to SCI commands based on how powerful the command is, and how knowledgeable each user is. The default privilege level is zero.

The optional field [,<prompt list>] can follow the privilege level definition. The comma is placed at the end of the .PROC line to indicate continuation of the line, if the prompt list is on the following line. The format of the field prompt list is as follows:

```
prompt[=type]
```

In the example, the prompt is the character string that you want to appear on the screen as the prompting message. The type is optional, and represents the name of one of the field prompt types discussed earlier. If a field prompt type is not specified, the prompting message is not displayed, and the prompt is called a "hidden prompt". More than one prompt can be specified by entering them in a list, separated by commas. Do not place a comma after the last item in the field prompt list. An example of a field prompt list is shown below:

```
.PROC EXP(EXAMPLE PROC)=0,
          INPUT FILE = ACNM, OUTPUT FILE =
          ACNM(INITIAL), LINES PER PAGE = INT(55),
          NUMBER OF COPIES = INT(1)
          . . .
.EOP
```

The preceding example would prompt the user to enter four fields: INPUT FILE, OUTPUT FILE, LINES PER PAGE, and NUMBER OF COPIES. The equal sign and the comma both function as continuation characters. The NUMBER OF COPIES prompt line has no comma after it.

A maximum of 22 prompts can be displayed on the 911, 931, 940, and Business System terminal screens, and up to 10 prompts can be displayed on the 913 VDT.

6.3.7.2 .BID, .QBID, .DBID, and .TBID Primitives. The .BID, .QBID, .DBID, and .TBID primitives each place a specified task into execution. There are several other ways to place a task into execution, as explained in Section 2. However, using primitives to execute a task allows the task to access the SCI PARMS list, which is the record in S$FGTCA and S$BGTCA containing all synonyms. The PARMS list allows you to pass values from the command procedure to the task.

The .BID primitive places a task into execution in the foreground mode of execution. Foreground execution means that SCI closes the LUNO assigned to the initiating terminal, then suspends itself until the executing task has terminated. This allows the executing task full access to the terminal. All synonyms in the terminal communications area are available to and shared by all tasks executing in foreground. Foreground tasks execute serially, so you cannot invoke other tasks at the terminal until the task bid has completed, except with .TBID.

.BID. The .BID primitive has operands to identify the command processor and pass parameters to the processor.

The .BID primitive requires the following format:

```
.BID TASK = < name/int> [,LUNO = < int> ][,CODE = < int> ]
     [,PARMS = (< string,...,string> )]
```

The TASK = name/int parameter identifies the task to be bid. The identifier can be either a name or an installed ID. The specified task must currently reside on a program file, and the LUNO parameter identifies a LUNO assigned to that program file. For .BID, the LUNO may be either station local or global. For .QBID and .DBID, it must be a global LUNO. You must specify a LUNO if the program file where the task resides is not the system program file. The default value for the LUNO is zero, and indicates the .S$PROGA program file. You can assign the LUNO to the appropriate file any time before you execute the task. Use the AL or AGL commands manually or in the IS procedure, or, you can assign the LUNO in the batch stream that initiates the task execution.

The CODE parameter represents the termination code of the executing task. This code can be accessed by the executing command processor through an S$STAT interface subroutine call. You can assign an integer value 0 through 255 to CODE; the default value is zero.

PARMS is a list of character strings, separated by commas, that can be accessed by the task. PARMS is used to supply any required parameters needed by the task being bid. The parameters specified here are accessed through a S$PARM subroutine call, and are not the same as parameters obtained by the Get Parameters SVC.

SCI transfers control to the task when it encounters the .BID. When the task terminates, SCI processes the next statement in the command procedure. The following is an example of the .BID primitive usage:

```
.PROC    EXP(EXAMPLE PROC — BIDS LIST DIRECTORY) = 0,
         PATHNAME = *ACNM(@$LD$P),
         LISTING ACCESS NAME = *ACNM
.SYN     $LD$P = "&PATHNAME"
.IF      "&PATHNAME",NE,""
.BID     TASK = > 32,CODE = > 24,
         PARMS = (> 12,@&PATHNAME,@&LISTING ACCESS NAME)
.ENDIF
.EOP
```

In the preceding example, the .BID primitive is used to bid a task with an ID of > 32 residing on the .S$PROGA program file. Three parameters passed to the > 32 task are responses to prompts.

Literals and synonyms may also be passed as parameters. Commas are used to separate parameters. The task initiated by the .BID primitive uses interface subroutines to access the parameters. These subroutines are discussed in detail later in this section.

When the .BID primitive bids a task, the synonym $$CC is set to 0 before the task begins executing. When the task completes, any value coded in the call to S$STOP gets placed in $$CC, and is passed to the foreground as the completion code. (The $$CC synonym is discussed further with the .STOP primitive.)

.QBID and .DBID. .QBID and .DBID use the same format as the .BID primitive, except any LUNO specified for the LUNO keyword must be global. When .QBID or .DBID is used to bid a task, SCI does not suspend, and does not close the LUNO assigned to the terminal. Therefore, the bid task should not attempt to send output to the terminal.

Both .QBID and .DBID bid tasks for execution in background. In background execution, a copy of the foreground synonyms is made for the exclusive use of the task executing in background, since tasks running in background cannot access foreground synonyms. This is necessary to protect synonyms used by background tasks from being inadvertently modified by the operator.

None of these synonyms copied for background use can be passed back to foreground except the $$BC synonym, which carries the completion code. (This completion code is supplied in the CODE parameter discussed earlier.) The S$STOP interface routine or the .STOP primitive passes the code to the command procedure. It is not set until control is returned to the primary input, so you cannot test it for the value of the return code until then. Primary input is the command control stream. For foreground input, this is the SCI prompt ([ ]), in background it is the batch stream. The synonym $$CC also gets assigned the completion code value, but it has no effect since it is not passed back to foreground.

The .DBID primitive bids a task for background execution and immediately suspends it. The task does not begin execution until it is activated, usually by one of the debug commands associated with the Debugger, described in Section 11.

.TBID. The .TBID primitive bids a task and terminates SCI at the associated terminal. SCI must be activated when the task terminates. If SCI is in background (batch), .TBID executes the task in background. If SCI is in foreground, .TBID executes the task in foreground. When .TBID bids a task in foreground, any background task continues to run.

Since SCI is terminated after .TBID is issued from a command procedure, any statements in the procedure following .TBID are not executed. This is also true of a batch stream that issues .TBID.

6.3.7.3 .DATA and .EOD Primitives. Data may be copied to a file directly from an input stream by the .DATA primitive, which has the following format:

```
.DATA <acnm>[,EXTEND = <YES/NO>][,SUBSTITUTION = <YES/NO>]
      [,REPLACE = <YES/NO>]
```

NOTE

> SCI does not map the .EOD primitive from lowercase to uppercase. Therefore, the .EOD primitive must be issued in uppercase to terminate the data stream.

The <acnm> parameter must be a valid file name, but it need not already exist. The .DATA primitive copies data to the file specified by acnm. The user has three parameters (EXTEND, SUBSTITUTION, and REPLACE) which affect the copying process.

The EXTEND parameter specifies whether or not the data file is to be opened extended. This parameter permits the user to concatenate several data streams under one pathname. The default value of EXTEND is NO.

The SUBSTITUTION parameter specifies whether textual substitution is to be done on the data stream before it is copied to the specified file. A response of YES causes the appropriate values to be substituted for indicated field prompts and synonyms, and multiple blanks compressed to a single blank unless enclosed by quotes. The default value of SUBSTITUTION is NO.

The REPLACE parameter specifies whether or not the data stream is to replace the file, if it already exists. The default value of REPLACE is YES.

The following is an example procedure that uses the .DATA and .EOD primitives:

```
.PROC EXP(EXAMPLE PROC)=0,
      INPUT PATHNAME=ACNM(@$EX$IP),
      OUTPUT PATHNAME = (ACNM)
      CC IAN = @&INPUT PATHNAME,
          OAN = @&OUTPUT PATHNAME
.DATA VOL1.MYLIB.MESSAGE,SUBSTITUTION=YES
COPY COMPLETED FOR:   @&INPUT PATHNAME
                           TO:   @&OUTPUT PATHNAME
.EOD
.EOP
```

In this example, the .DATA and .EOD primitives are used to write a message to the VOL1.MYLIB.MESSAGE file after an input file has been copied to a specified output file. If the user response to the INPUT PATHNAME prompt is VOL1.MYFILE.IN and the response to the OUTPUT PATHNAME prompt is VOL1.MYFILE.OUT, the message written to the MESSAGE file is:

```
COPY COMPLETED FOR: VOL1.MYFILE.IN
TO: VOL1.MYFILE.OUT
```

The Copy/Concatenate (CC) command is used in the example to copy the input file to the output file. The message written to the file VOL1.MYLIB.MESSAGE shows the input and output file names. IAN and OAN are prompt names formed according to the rules of abbreviation previously explained. IAN refers to INPUT ACCESS NAME(S) and OAN refers to OUTPUT ACCESS NAME, the required prompts in the CC command procedure. These are examples of abbreviated field prompt references.

Refer to the description of the .IF primitive for a more detailed explanation of using procedure calls within procedures.

6.3.7.4 .EVAL Primitive. The .EVAL primitive has the following format:

```
.EVAL <synonym> = <value>
```

The .EVAL primitive evaluates a string as a numeric expression, converts the result to decimal ASCII, and stores it as the value of a synonym. The string to be evaluated is specified in the <value> parameter. The following example assigns the value 13 to the synonym RESULT.

```
.SYN THREE = 3
.SYN TWO = 2
.SYN RESULT = 0
.EVAL RESULT = @RESULT+@THREE*5-@TWO
```

The .EVAL primitive provides the arithmetic capability of SCI. It also provides a mechanism for setting up counter variables, as shown in the example for the .LOOP primitive.

NOTE

> The RESULT synonym must not have the at sign (@) preceding it on the left of the equal sign. If the same synonym is used on the right side of the equal sign, it must have the at sign (@).

If prompt names are used in a .EVAL primitive only one prompt name may appear to the right of the equal sign and that must be the last name on the line. For example:

```
.EVAL RESULT = @RESULT+&NO OF COPIES?
```

If the prompt &NO OF COPIES? is the first operand after the equal sign, followed by a plus sign (+), the plus sign is interpreted as a part of the prompt.

6.3.7.5 .EXIT Primitive. The .EXIT primitive is used to terminate the execution of a current command procedure. (The .EOP is used to terminate the definition of a command.) The .EXIT can be used anywhere within a command procedure definition, as often as needed.

The .EXIT primitive has the following format:

```
.EXIT
```

The following is an example of the .EXIT primitive:

```
.PROC EXP(EXAMPLE PROC)=0,
     INPUT PATHNAME=ACNM(@$EX$IP),
     DELETE FILE?=YESNO(N)
     SF FILE=&INPUT PATHNAME
     .IF &DELETE FILE?,NE,"Y"
     .EXIT
     .ENDIF
        DF PATHNAME = &INPUT PATHNAME
.EOP
```

In this example if &DELETE FILE? is not equal to Y, the .EXIT primitive terminates execution of the procedure. If &DELETE FILE? is equal to Y, the input file is deleted.

6.3.7.6 .IF, .ELSE, and .ENDIF Primitives. The conditional primitive of the SCI language is the conventional IF-THEN, IF-THEN-ELSE construction. The .IF conditional primitive is used in conjunction with the .ELSE and .ENDIF conditional primitives. The .IF and .ELSE primitives allow the user to specify an action depending on the outcome of a comparison. The .ENDIF primitive terminates the .IF primitive.

The .IF primitive has the following format and must be used in conjunction with the .ENDIF and possibly the .ELSE primitives:

```
.IF     <op1>,<relation>,<op2>
   .
   .
   .
.ELSE
   .
   .
   .
.ENDIF
```

When the .IF condition is true, statements following the .IF are executed. When the condition is false, the statements following the .ELSE, if present, are executed. Execution then continues with the statements after the .ENDIF.

The relation between the op1 and op2 parameters are as follows:

Relation	Meaning
<op1>,EQ,<op2>	op1 is equal to op2
<op1>,NE,<op2>	op1 is not equal to op2
<op1>,GT,<op2>	op1 is greater than op2
<op1>,LT,<op2>	op1 is less than op2
<op1>,GE,<op2>	op1 is greater than or equal to op2
<op1>,LE,<op2>	op1 is less than or equal to op2

The op1 and op2 parameters may be strings, variables, or concatenated strings. If both op1 and op2 are numeric, a numeric comparison is done. Otherwise, a string comparison is done in the ASCII collating sequence.

Any SCI primitives or calls to other procedures may be used between an .IF and an .ENDIF primitive including another .IF primitive. Nested conditionals are allowed up to 32 levels deep.

The .ELSE primitive is used with the .IF and .ENDIF primitives to allow the user to specify an action to be executed when the .IF comparison yields a false condition.

The .ENDIF conditional primitive terminates the .IF primitive.

The following example shows the use of the .IF, .ELSE, and .ENDIF primitives. In the example, IAN and OAN of the CC command represent the input and output pathnames, respectively. The prompt names supplied to the CC procedure are abbreviations of INPUT ACCESS NAME(S) AND OUTPUT ACCESS NAME.

```
.PROC EXP(EXAMPLE PROC)=0,
     INPUT PATHNAME=ACNM,
     OUTPUT PATHNAME=ACNM,
     DELETE?=YESNO(N)
.IF &DELETE?,NE,"N"
       .IF&DELETE?,NE,"Y"
         MSG TEXT = "RESPONSE TO DELETE? MUST BE Y OR N"
         .EXIT
       .ENDIF
.ENDIF
CC IAN = &INPUT PATHNAME,
    OAN = &OUTPUT PATHNAME
.IF &DELETE?,EQ,-"Y"
       DF PATHNAME(S) = &INPUT PATHNAME
.ENDIF
.EOP
```

In this example, the .IF primitive is used to compare the user response to the DELETE? prompt to Y or N. If the response is any value other than Y or N, the message is displayed and the procedure returns to SCI via the .EXIT primitive.

After verifying that the value of DELETE? is either Y or N, the procedure copies the input file to the output file. After the copy, the input file is deleted using the DF (Delete File) command procedure if the response to DELETE? was Y.

6.3.7.7 .LOOP, .UNTIL, .WHILE, and .REPEAT Primitives. The loop primitives (.LOOP, .UNTIL, .WHILE, .REPEAT) are used to repeat blocks of SCI statements. The .LOOP primitive begins the repetition, which must be terminated by a .REPEAT primitive. The .UNTIL or .WHILE primitives can be used anywhere and as many times as necessary in the repeated block.

The loop primitives have the following format:

```
.LOOP
.UNTIL <op1>,<relation>,<op2>
.WHILE <op1>,<relation>,<op2>
.REPEAT
```

The op1 and op2 parameters may be strings, variables, or concatenated strings. The relation parameter designates the type of string or numeric comparison to be performed. The basic structure of a loop in an SCI procedure is as follows:

```
.LOOP
  .
  .      SCI statements
  .
.UNTIL   or .WHILE
  .
  .      SCI statements
  .
.REPEAT
  .
  .
  .
```

The loop is initiated by the .LOOP primitive and ended by a .REPEAT primitive. The loop must contain at least one .WHILE or .UNTIL primitive, and can contain more than one such primitive. Both the .WHILE and .UNTIL primitives can occur anywhere within the loop. The SCI statements within the loop are continually executed until the condition specified by the .WHILE primitive becomes *false*, or the condition specified by the .UNTIL primitive becomes *true*, at which time SCI executes the first statement following the .REPEAT primitive. Loops can be nested to a maximum depth of 32.

If multiple .UNTIL and .WHILE primitives are contained within a loop, SCI will discontinue the loop when any .UNTIL condition becomes true or any .WHILE condition becomes false. SCI then executes the first statement following the .REPEAT.

The following is an example of the loop primitives.

```
.PROC EXP(EXAMPLE PROC) = 0,
      INPUT PATHNAME = ACNM(@$EX$IP),
      LISTING DEVICE = NAME(@$PF$D),
      NUMBER OF COPIES? = INT(1)
      .SYN $EX$IP = &INPUT PATHNAME
      .SYN $PF$D = &LISTING DEVICE
      .SYN NUM = &NUMBER OF COPIES?
.LOOP
      .UNTIL @NUM,LE,0
      PF       FILE = @$EX$IP,L = @$PF$D
      .EVAL    NUM = @NUM - 1
.REPEAT
      .SYN NUM = ""
.EOP
```

In this example, the .LOOP, .UNTIL, .EVAL, and .REPEAT primitives function together as a counting mechanism.

The synonym NUM is given the integer value entered as the response to the NUMBER OF COPIES? prompt. The .UNTIL primitive checks the value of NUM against zero. If NUM is greater than zero, the PF command procedure is bid to print the file identified by the response to the INPUT PATHNAME prompt on the listing device identified by the response to the LISTING DEVICE prompt. Then, the .EVAL primitive decreases the value of NUM by 1, and the .REPEAT primitive causes the loop to repeat. The loop repeats until the value of NUM is less than or equal to zero, at which time execution continues with the primitives following the .REPEAT primitive.

The .SYN primitive at the end of the procedure is used to delete the synonym NUM from the synonym table. Procedures must delete internal synonyms before terminating, or the synonym table can overflow.

It is possible to enter a negative value in response to the INT prompt type. This is the reason for comparing less than or equal to zero in the .UNTIL primitive.

An infinite loop is easily created when using the .LOOP primitive, if the .WHILE or .UNTIL are improperly coded. If such a loop occurs, you cannot escape using the terminal control keys. If the system has only one terminal, the solution is to reinitialize the system. However, if there are several active terminals, this solution may be impractical. In this case enter the Show Task Status (STS) command at another terminal to obtain the RUN ID of the task (you must know the station ID of the terminal affected). Then issue a Kill Task (KT) command against that ID to abort it.

6.3.7.8 .MENU Primitive. The .MENU primitive causes SCI to display a specified menu when SCI returns to command mode. A menu cycle occurs just before the SCI prompt is displayed. The .MENU primitive only affects the next menu cycle. Subsequent menu cycles revert to the default menu. The .MENU primitive has the following format:

.MENU [< menu name>]

There are three variations of the menu name parameter:

- No menu specified — Use of a .MENU with no menu name specified causes SCI to bypass the menu cycle before the next SCI prompt is displayed.
- Menu name — If a menu name (1 through 6 alphanumeric characters) is specified, SCI displays the menu in the next menu cycle, whether the station is in TTY or VDT mode. SCI appends the characters M$ at the beginning of the menu name to obtain the file name within the user's command procedure library(s) of the file where the menu resides.
- Menu name — If a menu name preceded by an * is specified, the menu is displayed only if the station is in VDT mode.

The slash (/) symbol, entered in response to the SCI prompt, is converted by SCI to .MENU. The standard menu shown later in this section illustrates the use of the slash. The terminal user enters /PDEV to see the program development menu. The slash is resolved by SCI to .MENU and the entry becomes .MENU PDEV.

/	is equivalent to .MENU
/ DEV	is equivalent to .MENU DEV
/*DEV	is equivalent to .MENU *DEV

The following is an example of the .MENU primitive:

```
.PROC    NM(NEW MENU PROC)
           .MENU MYMENU
.EOP
```

The menu shown with the .MENU command displays once. To change the default menu displayed with the SCI prompt, use the MENU parameter of the .OPTION primitive. The standard system menu is named .S$PROC.M$LC.

6.3.7.9 .OPTION Primitive. The .OPTION primitive enables users to modify some basic interface characteristics of SCI to suit local language or application requirements. The .OPTION primitive has the following format:

```
.OPTION [PROMPT = < string> ][,MENU = < name> ][,PRIMITIVES = < YES/NO> ]
        [,LOWERCASE = < YES/NO> ]
```

The parameter definitions are as follows:

Keyword	Assigned Value	Function
PROMPT	An alternative prompt character string, which must be less than 50 characters in length.	Enables you to specify the SCI prompt. The default SCI prompt is [ ], represented by the ASCII codes of >7B and >7D.
MENU	Main menu name. The M$ prefix is automatically supplied by SCI for the specified menu name, so the file can be located in the user directory.	Allows you to control which top-level menu is displayed when control returns to SCI.
PRIMITIVES	YES or NO, with YES as the default.	Prevents interactive use of primitives. If you specify .OPTION PRIMITIVES = NO, you cannot enter a primitive in response to the SCI prompt [ ]. Primitives are still allowed in procedures. Also, .OPTION PRIMITIVES = NO in a batch stream prevents further use of primitives in that batch stream.
LOWERCASE	YES or NO, with NO as the default.	Enables or disables lower case to upper case mapping on input to SCI.

NOTE

> The LOWERCASE option applies only to prompts processed by SCI directly. Responses to prompts processed within a command processor cannot be mapped to uppercase, since they are not directly under SCI control. Examples of commands in which the lowercase option does not apply are: DCOPY, MS, XANAL, MVI and INV. Once invoked, these commands bypass the SCI interface, and perform their own input processing. The user ID and password supplied during log on must also be entered in uppercase.

The following example shows the use of the .OPTION primitive to display the characters [YES, CAPTAIN?] instead of the usual SCI prompt, [ ].

```
.PROC EXP(NEW PROMPT PROCEDURE)
        .OPTION PROMPT = "[YES, CAPTAIN?]"
.EOP
```

Alternative prompts can also be displayed, depending on the user ID in use at a terminal, by writing the following .OPTION primitive in the M$00 file of the command procedure library:

```
.OPTION PROMPT = ST@$$ST USER:@$$UI[ ]
```

If the synonym $$UI contained the value JF0012, the following text would display when the user logs on:

```
ST01 USER:JF0012[ ]
```

The synonyms in the preceding example are only evaluated at the time the .OPTION primitive is executed. The values of the synonyms at that time become the permanent values of the SCI prompt until another .OPTION statement is executed or a log-off, log-on sequence is performed for the terminal.

In the following example, the .OPTION primitive is used to select M$EDIT menu for display at each menu display cycle of SCI. This command also disables primitives at the primary level, and enables the use of lower case characters as input to SCI.

```
.OPTION MENU = EDIT, PRIMITIVES = NO, LOWERCASE = YES
```

6.3.7.10 .OVLY Primitive. The .OVLY primitive is used in standard DX10 SCI procedures. It is documented here to help you understand the function of the SCI procedures that use it.

The .OVLY primitive overlays portions of the SCI code in certain procedures where the command processor is quite lengthy.

Using the .OVLY primitive to call existing SCI functions is not recommended, since references to this primitive may not be compatible with future extensions of the operating system. Further, there is a considerable potential for system failure due to improperly coded .OVLY calls.

The .OVLY primitive is not suitable to call user-defined functions, since you cannot properly link such functions with the overlay mechanism. The .OVLY primitive invokes overlays to the SCI code itself, and is therefore restricted to functions previously linked to SCI. The DX10 object kit does not contain all the elements necessary to link SCI.

CAUTION

Improper use of the .OVLY primitive can result in system failure.

6.3.7.11 .PROMPT Primitive. Use the .PROMPT primitive to solicit additional information from a terminal operator. The .PROMPT primitive can be used to collect additional prompts when more are required than can be displayed with the .PROC prompt list. When using the .PROC prompt list, the 911, 931, 940, and Business System terminals can display up to 22 prompts; the 913 VDT can display a maximum of 10 prompts. The .PROMPT primitive can be used to collect optional responses after normal procedure prompts. The syntax for the .PROMPT primitive is as follows:

```
.PROMPT [(<full name>)][=<int>][,<prompt list>]
```

The prompts defined by .PROMPT are not displayed on the same screen with those defined by .PROC. The screen is cleared and the new prompts are displayed. For a procedure with .PROMPT in it to be usable in batch, all the prompts required by .PROMPT primitives must be named as hidden prompts as part of the .PROC prompt list.

The full name parameter is optional and is a character string to be displayed when .PROMPT is executed in interactive mode. The <int> parameter specifies the lowest privilege level that can invoke the procedure. However, this privilege level can be higher than that specified by .PROC. The <prompt list> parameter is a list of the prompts to which the user responds.

The following is an example of the .PROMPT primitive:

```
.PROC EXP(EXAMPLE PROC)=0
        INPUT PATHNAME=ACNM,
        OUTPUT PATHNAME=ACNM,
        DISPLAY OR COPY?=STRING(DISPLAY)
        .IF &DISPLAY OR COPY?,EQ,"DISPLAY"
         SF FILE=&INPUT PATHNAME
         .ELSE
         CC IAN=&INPUT PATHNAME,
             OAN=&OUTPUT PATHNAME
.PROMPT (SUPPLEMENTARY QUESTION)=3
        DELETE FILE?=YESNO(N)
        .IF &DELETE FILE?,EQ,"Y"
         DF PATHNAME=&INPUT PATHNAME
         .ENDIF
         .ENDIF
.EOP
```

In this example, the DELETE FILE? prompt of the supplementary questions is not displayed unless the user is copying the file and the procedure is invoked by a user whose privilege level is at least 3. A user with a privilege level lower than 3 receives an error message when the .PROMPT is encountered in the procedure, and the procedure is aborted at that point.

6.3.7.12 .SHOW Primitive. The .SHOW primitive displays the contents of a specified file, or files, to an interactive terminal. The .SHOW primitive has the following format:

```
.SHOW <acnm>[,<acnm,...>]
```

Here, acnm is the name of a file or a synonym referring to a file.

The .SHOW primitive cannot be used to show a program file or an image file. .SHOW is the equivalent of issuing an SF command.

The following is an example of the .SHOW primitive:

```
.PROC EXP(EXAMPLE SHOW FILE) = 0,
          INPUT FILENAME = ACNM(@$SF$P)
          .SYN $SF$P = &INPUT FILENAME
.SHOW @@$SF$P
.EOP
```

In this example, the .SHOW primitive causes the file identified by the response to the INPUT FILENAME prompt to be displayed. The initial value of the prompt is set to the file name of the last input file. The name of the last input file is then changed by the .SYN primitive to the name of the file entered as INPUT FILENAME to this procedure.

Synonyms used as file name prompts for the .SHOW primitive must be coded with two at signs (@@) as shown in the example. If a prompt name is used and a synonym name can be entered in the prompt, the prompt name must be preceded by one at sign (@). For example, @&INPUT FILENAME indicates that a synonym can be used for the input file name.

6.3.7.13 .SPLIT Primitive. The .SPLIT primitive is used to remove the first term from a value list and has the following format:

```
.SPLIT LIST = < string> , FIRST = < synonym name> [,REST = < synonym name> ]
```

The < string> supplied on the right-hand side of the LIST = must begin with a left parenthesis and end with a right parenthesis. The operation of the .SPLIT primitive is best explained by the following examples:

Before Execution of the .SPLIT Primitive		Results After Execution of the .SPLIT Primitive	
Syntax of LIST	**Value of Synonym in Example Syntax**	**Value of Synonym on Right-Hand Side of FIRST =**	**Value of Synonym on Right-Hand Side of REST =**
LIST = (A,B,C)	—	A	(B,C)
LIST = (A)	—	A	null
LIST = ( )	—	null	null
LIST = ((X,Y),Z,G)	—	(X,Y)	(Z,G)
LIST = (@SYN)	A,B,C	A	(B,C)
LIST = @WXB	(A,B,C)	A	(B,C)
LIST = (@XYZ)	(A,B,C)	(A,B,C)	null
LIST = (@B1)*	A(,B,C)		
LIST = @B2*	A,B,C		

Note:

* Produces an error.

Items in the value list must be separated by commas. Parentheses are used to control how the list is split.

The following is an example of the .SPLIT primitive used in a command definition:

```
.PROC EXP(EXAMPLE PROC),
      INPUT PATHNAME = ACNM(@$EX$IP),
      OUTPUT PATHNAME(S) = (ACNM)
      .SYN $EX$IP = &INPUT PATHNAME
      .SYN $EX$OP = (&OUTPUT PATHNAME)
      .LOOP
        .SPLIT LIST = @$EX$OP,
        FIRST = $EX$P,
        REST = $EX$OP
        CC IAN = @$EX$IP,OAN = @$EX$P
      .WHILE @$EX$OP,NE,$EX$OP
      .REPEAT
.EOP
```

In this example, the file identified by the response of the INPUT PATHNAME prompt is copied to the file, or files, identified by the responses to the OUTPUT PATHNAME(S) prompt. The .SPLIT primitive is used within a loop to access the current output file to which the input file is to be copied. When one copy has completed, .SPLIT updates the current output file to be the next output file.

The .WHILE primitive is used within the loop to check if the input file has been copied to all specified output files. When the synonym EXOP has no value, the output file pathnames have been exhausted.

The parentheses around &OUTPUT PATHNAME in the .SYN primitive are required for a list value.

The .WHILE primitive should be encoded as shown with an at sign (@) on the first synonym name and no at sign on the second synonym name. When the first operand, @EXOP, resolves to a null value because the list is empty, the resolved value becomes the synonym name, EXOP. The second operand in the .WHILE primitive is treated as a literal because it has no at sign in front of it. The two operands are equal and the loop terminates at this point.

6.3.7.14 .STOP Primitive. The .STOP primitive in a foreground procedure logs the user off SCI. In a batch stream, .STOP ends the batch stream. .STOP has the following format:

```
.STOP [TEXT=<string>][,CODE=<int>]
```

The string specified by the TEXT parameter and the CODE value are optional. The TEXT parameter can be used to pass a string back to the foreground SCI to be displayed instead of the BATCH SCI HAS COMPLETED message. The TEXT = parameter cannot exceed a single line in the procedure.

During execution of a foreground task, the CODE parameter can be used to set the synonym $$BC in the synonym table of the bidding SCI task to a four-digit hexadecimal value at the completion of the batch stream.

When a background task is initiated, the foreground synonym table of the SCI task at that terminal is copied to background for use by the background task. At this time, the $$BC synonym is set to null. The background task can place the completion code in $$BC and return it to foreground in the the CODE parameter.

The value is not placed in the $$BC synonym until any active foreground procedure terminates. Therefore, the $$BC synonym can only be used by a procedure executed after the procedure originally calling the background task. SCI ignores the TEXT and CODE parameters if not in batch mode.

If the .STOP primitive is issued in foreground mode, any unfinished text editing is lost.

The following is an example of the interactive use of the .STOP primitive:

```
.PROC EXP(TERMINAL IS LOGGING OFF)=0        !FULL NAME DISPLAYED
        .STOP                               !TERMINATE SCI
.EOP
```

In this example, the .STOP primitive is used within a command procedure to stop SCI and log-off the terminal. This is functionally equivalent to the Q command used to quit SCI.

Upon termination of a batch stream, SCI passes the message specified in the TEXT parameter back to the foreground SCI for the terminal from which the batch stream was submitted. If the terminal is in a WAIT command, SCI immediately displays the TEXT message. If the terminal is displaying the SCI prompt, pressing the Return key causes a display of the message. If the terminal is processing a command in foreground mode, the message will be displayed when the foreground task is completed.

6.3.7.15 .SYN Primitive. The .SYN primitive is used to assign values in the synonym table and has the following format:

```
.SYN <name> = "<value>"...
```

The <name> parameter specifies the name of a synonym without the at sign (@). The <value> can be a string, variable, or a concatenated expression. Parentheses are required for values containing lists.

Assigning a null value to a synonym deletes the synonym from the synonym table. Assign a null value to a synonym as follows:

```
.SYN <name> = ""
```

The following is an example of the .SYN primitive used in a command definition:

```
.PROC   EXP(EXAMPLE PROC)=0,
         INPUT PATHNAME(S)=(ACNM)(@$EX$IP),
         OUTPUT PATHNAME=ACNM(@$EX$OP),
.SYN  $EX$IP=(&INPUT PATHNAME)
.SYN  $EX$OP=&OUTPUT PATHNAME
        CC IAN=@$EX$IP,
           OAN=@$EX$OP
.EOP
```

This example assigns values to synonyms EXIP and EXOP. These synonyms may be accessed by other command procedures. The synonym EXIP is the name of the last input file and EXOP is the name of the last output file. Procedures can assign values to these synonyms, making them accessible to subsequent SCI tasks.

6.3.7.16 .USE Primitive. The .USE primitive specifies the procedure directories, or libraries, to be used by SCI. The .USE primitive has the following format:

```
.USE [<acnm1>][,<acnm2>][,<acnm3>][,<acnm4>][,<acnm5>]
```

After the .USE statement is executed, any command procedures or menus to be executed are searched for in the directories specified as <acnm1> through <acnm5>, in that order. The .USE primitive remains in effect until overridden by another .USE or until a log-off/log-on sequence occurs. To revert to the standard system library, specify a .USE with no operands. This causes the default value .S$PROC to be <acnm1>, and all other access names are null.

When a command procedure is installed using the .PROC primitive interactively or in a batch stream, SCI places the command definition into the directory specified by pathname 1.

One of the pathnames must contain the main menu specified by the .OPTION primitive or a warning message will occur after the .USE primitive is invoked. This menu is usually the file named .S$PROC.M$LC. When an SCI command is typed in response to the prompt [ ], SCI forms the file name of the command library by first concatenating the procedure name entered by the user to pathname 1, and if not found, concatenating the procedure name to pathname 2 and so on through pathname 5. If no file is found in any of the directories, you receive an error. The following prompts cause SCI to look for a command file named .S$PROC.EX:

```
[ ] .USE.S$PROC, .USERLIB
[ ] EX
```

If no such file is found, SCI then searches for .USERLIB.EX. The .USE command establishes the procedure library directory names.

NOTE

> If the default menu cannot be found after the .USE primitive is invoked, a warning is displayed and no menu is shown. If this occurs in a procedure, the procedure is aborted at that point. One of the procedure libraries named in the .USE command should have a file containing the default menu.

Issuing a .USE without parameters returns control to the system procedure library (.S$PROC).

The S$PROC.M$00 file is frequently used to invoke user procedure libraries with the .USE primitive. For example,

```
.PROC M$00
  .USE .S$PROC,.USERLIB
  .IF @$$UI,EQ,JF0012
   .USE .S$PROC,.JFLIB
   .OPTION MENU = JF
  .ENDIF
.EOP
```

In the preceding example, M$00 adds the procedure library .USERLIB for all users except the user with an ID of JF0012. JF0012 would use the procedure library .JFLIB and would also have a default menu name of M$JF.

If the M$00 procedure exists in the .S$PROC command directory, it is automatically executed when any user logs on to SCI. You can use M$00 to perform any custom initialization for your application needs, as discussed earlier in this section.

6.3.8 Processor Interfacing Subroutines

The SCI processor subroutines provide the interface between procedures and processors, provide data manipulation facilities, and control messages directed to the terminal user. These routines reside in .SCI990.S$OBJECT and can be linked with user programs.

When linking a program that uses these routines, it is usually best to let routine references be resolved through automatic symbol resolution using the LIBRARY statement in the link edit control stream. If this method is used, the LIBRARY .SCI990.S$OBJECT statement should be the first LIBRARY statement in the link stream.

The four categories of interface subroutines and the subroutines in each category are as follows:

- String Utility Subroutines
  - S$IASC — Convert Binary Integer to ASCII
  - S$INT — Convert ASCII to Binary Integer
  - S$SCOM — Compare Two Strings
  - S$SCPY — Copy String
- SCI Interface Subroutines
  - S$GTCA — Get Terminal Communications Area (TCA)
  - S$PTCA — Put TCA
  - S$BIDT — Bid a Task from the Processor
  - S$RTCA — Release TCA
  - S$NEW — Initialize the TCA
  - S$PARM — Get the I-th Parameter
  - S$SETS — Set Synonym Value
  - S$MAPS — Map Synonym (get its value)
  - S$SNCT — Search Name Correspondence Table
  - S$STAT — Get Terminal Status
  - S$SPLT — Split List into Components
  - S$STOP — Return to SCI

- Arithmetic Utility Routines
  - S$IADD — Add 32-bit Integers
  - S$ISUB — Subtract 32-bit Integers
  - S$IMUL — Multiply 32-bit Integers
  - S$IDIV — Divide 32-bit Integers
- Local Display File Routines
  - S$OPEN — Open the Terminal Local File (TLF)
  - S$WRIT — Write to the TLF
  - S$WEOL — Write End-of-Line to the TLF
  - S$CLOS — Close the TLF

Each of these interface subroutines are described in the following paragraphs.

6.3.8.1 String Utility Subroutines. Operands for command processors are passed from SCI as character strings. These strings are stored in buffers with the following form:

STRING LENGTH	C1	C2	• • •	CN

2283183

The string length is the number of characters in the string (one byte per character), and the buffer is one byte longer than the string. A maximum of 255 can be specified as the string length. In the format illustration, C1, C2 and CN are characters in the string.

Empty buffers reserved for string storage should have the buffer length minus one (maximum string length) in the first byte. The following routines are provided to operate on strings.

S$IASC — Convert Binary Integer to ASCII. This routine is used to convert a 32-bit binary integer into an ASCII text string representing that number. R1 contains the address of the 32-bit number. R2 points to a buffer which is to receive the ASCII text. The buffer is of the form of the string variable, with the first byte containing the length of the buffer minus one. The 32-bit integer will either be taken as a *two's complement* number or as a positive binary number, depending on the base byte (second byte) of R3. If the base byte is zero, the number is treated as a *two's complement* binary integer and will be converted into the ASCII representation of the decimal (base 10) number, with leading blanks and a "-" character if the number is negative. If base is anything but zero, the 32-bit integer is treated as positive and will be converted into the ASCII representation of the integer in the specified base, with leading zeros. S$IASC puts the length of the ASCII string into the first byte of the receiving buffer.

Calling sequence: BLWP @S$IASC

Registers Used:

- R0 — Error code returned by S$IASC
- R1 — Address of the 32-bit integer
- R2 — Address of the buffer which is to receive the ASCII text
- R3 — Byte 0: number of ASCII characters to be output (field width); zero means variable number; maximum is 32
  - Byte 1: Base (e.g., 10 or 16) into which the integer is to be converted, prior to representation in ASCII (0 = decimal)

Example:

```
INT      DATA   2,0          INTEGER = 128K
BUFFER   BYTE   15           LENGTH OF BUFFER
         BSS    15
         LI     R1,INT       RI = ADDRESS OF INT
         LI     R2,BUFFER    R2 = ADDRESS OF BUFFER
         LI     R3,>0010     FIELD WIDTH = VARIABLE
*                            BASE = 16
         BLWP   @S$IASC      CONVERT
```

S$INT — Convert ASCII to Binary Integer. This routine converts an ASCII test string representing an integer expression into a 32-bit binary value. The integer expression to be converted can contain the standard arithmetic operators +, −, *, and /. If the ASCII string contains a number beginning with > or 0, the number is assumed to be hexadecimal; otherwise, it is assumed to be in the base specified in workspace register R4.

Calling Sequence: BLWP @S$INT

Registers Used:

- R0 — Error code returned by routine
- R2 — Pointer to the ASCII string to be converted to an integer
- R3 — Pointer to a 4-byte (32-bit) buffer in which the converted binary integer is to be stored
- R4 — Base of the number represented by the input string (e.g., 10 or 16). If R4 is zero, base 10 is assumed.

Example:

```
NUMBER  BYTE   5            LENGTH OF TEXT
        TEXT   '>AE80'      HEX NUMBER
        EVEN
INT     BSS    4            BUFFER FOR INTEGER
        LI     R2,NUMBER    R2 = ASCII NUMBER
        LI     R3,INT       R3 = BUFFER
        LI     R4,16        R4 = BASE 16
        BLWP   @S$INT       CONVERT THE NUMBER
```

S$SCOM — Compare Two Strings. This routine compares two strings, and sets the arithmetic bits in the status register to reflect the results of the comparison. If one string is shorter than the other, it is assumed to be filled with the NULL characters (hexadecimal 00). If one string is a substring of the other (matching from the left), R0 is set to zero. The two strings are pointed to by registers Ra and Rb, where Ra and Rb are specified in the two bytes immediately following the call to S$SCOM (see example).

Calling Sequence:

```
BLWP @S$SCOM
BYTE Ra, Rb
```

Registers Used:

R0 — Substring test code returned by S$SCOM: If 0, one string is a substring. If −1, the strings differ at some character.
Ra — Pointer to the first string
Rb — Pointer to the second string

Example:

```
FIRST   BYTE   6            LENGTH OF FIRST STRING
        TEXT   'SUBSTR'
SECOND  BYTE   9            LENGTH OF SECOND STRING
        TEXT   'SUBSTRING'
        LI     R3,FIRST     R3 POINTS TO FIRST
        LI     R5,SECOND    R5 POINTS TO SECOND
        BLWP   @S$SCOM      COMPARE THE TWO
        BYTE   R3,R5        DEFINE 'A' AND 'B'
        MOV    R0,R0
        JEQ    SUB          THIS JUMP WILL OCCUR
```

S$SCPY — Copy String. This copies the string pointed to by register Ra into the buffer pointed to by register Rb, placing the length of the copy string in the first byte of Rb. Registers Ra and Rb are defined in the two bytes immediately following the call to S$SCPY. The buffer containing the string (pointed to by register Ra) must not overlap the buffer in which the copy is to be placed. The buffer at Rb must be set up as a string buffer, as described earlier in this section. If the length of the receiving buffer is less than the text string to be copied, an error code is returned in R0. If register Ra is zero, or the string to be copied is the null string (zero length), the buffer length (first byte) of the buffer at Rb will be set to zero.

Calling Sequence:

```
BLWP  @S$SCPY
BYTE  Ra, Rb
```

Registers Used:

R0 — Error code returned by S$SCPY
Ra — Pointer to text to be copied
Rb — Pointer to buffer to receive text

Example:

```
STRING  BYTE  7             LENGTH OF STRING
        TEXT  'COPY ME'
COPY    BYTE  20            LENGTH OF BUFFER
        BSS   20
        LI    R1,STRING     R1 = POINTER TO STRING
        LI    R8,COPY       R8 = POINTER TO BUFFER
        BLWP  @S$SCPY       CALL S$SCPY
        BYTE  R1,R8         DEFINE 'A' AND 'B'
          .
          .
          .
```

6.3.8.2 SCI Interface Subroutines. SCI interface subroutines allow a command processor to access parameters passed to it from the command procedure through the PARMS and CODE keyword values in the .BID, .QBID, .DBID, or .TBID primitve or an S$BIDT subroutine call. The task must be bid in one of these five manners to access the parameters. They also allow the command processor to access and modify synonyms defined for the teminal, and to return control to SCI. These synonyms reside in the terminal communications area (TCA) of the terminal, which acts as an information buffer between SCI and processors.

You must issue a S$GTCA subroutine to get the TCA before calling any of the other routines in this category. After your processing is complete, issue a S$RTCA subroutine call to release the TCA. Return control to SCI by issuing an S$STOP subroutine. Although you can issue the S$STOP routine at any time, it is good practice to release the TCA first.

S$GTCA — Get Terminal Communications Area. This routine makes the TCA available for use by the calling routine (a command processor). This routine must be called before a command processor can access prompt and synonym values, and receive parameters passed to it by the SCI procedure.

Calling Sequence: BLWP @S$GTCA

Registers Used: R0 — Error code returned by S$GTCA

Example:

```
*BEGINNING OF COMMAND PROCESSOR
BEGIN BLWP @S$GTCA OPEN TCA
  .
  .
  .
```

S$PTCA — Put Terminal Communications Area. This routine must be called by a command processor that has written in the TCA (using the S$SETS routine), before the processor terminates or calls S$RTCA.

Calling Sequence: BLWP @S$PTCA

Registers Used: R0 — Error code returned by S$PTCA

Example:

```
.
.
.
.
*THE TCA HAS BEEN MODIFIED
WRITE BLWP @S$PTCA UPDATE THE TCA
.
.
.
```

S$BIDT — Bid a Task. This routine allows you to bid tasks from within currently executing tasks. The task must be linked with these routines and must be bid with one of the five methods described in paragraph 6.3.8.2. (If it is not linked with these routines, you can execute another task using the Execute Task SVC (opcode >2B).)

Before calling S$BIDT, initialize the TCA with an S$GTCA call.

Calling Sequence: BLWP @S$BIDT

Registers Used:

R0 — Error code returned by S$BIDT

R1 — The left byte contains the task ID and the right byte contains the program file LUNO associated with the bid task.

R2 — Contains the address of a table of addresses pointing to parameters passed to the bid task. Set this register to zero to pass the parameters of the calling task to the bid task without modification.

R3 — Left byte is the CODE passed to the bid task. Set to zero if no CODE is passed. The right byte indicates flags in bits 8, 12, 15. Set other bits to zero. The flag bits indicate the following:

Bit 8 — Return runtime ID flag
Bit 12 — Terminate calling task flag
Bit 15 — Suspend calling task flag

The calling task places the ID of the bid task in the left byte of R1, and the LUNO of the bid task's program file in the right byte.

The table of addresses specified in R2 point to the parameters passed to the bid task. The table must contain zero as the last entry, and byte zero of each parameter must contain the number of characters in the parameter. Bytes 1 through n contain the characters of the parameter. To pass no parameters to the bid task, place an address in R2 that points to an address containing zero. This, in effect, is an empty table of addresses. If the table is not empty, your task must be small enough to allow S$BIDT to obtain 864 bytes of memory or an error is returned.

The calling task places the CODE parameter value in R3, and sets the flag bits as follows:

Bit 8	If set to 1, the calling task returns the runtime ID of the bid task in the left byte of R1. If set to 0, R1 remains unchanged.
Bit 12	If set to 1, the calling task is terminated after the called task is bid. If set to 0, the calling task remains active.
Bit 15	If set to 1, the calling task is suspended until the called task terminates. If set to 0, the calling task remains active.

NOTE

Either bit 12 or bit 15 must be set to 1 when the bid task is initiated. If both are zero, the results are unpredictable.

The following example sets up a structure to pass three parameters to a task:

```
ADRTBL  DATA PARM1
        DATA PARM2
        DATA PARM3
        DATA 0
*
PARM1   BYTE 3,'A','B','C'
PARM2   BYTE 0
PARM3   BYTE 2, '1','0'
```

The first parameter passed to the task is the string ABC, which is 3 bytes long. The second parameter is a null parameter (no value). The third parameter is the string 10, which is 2 bytes long. The table ADRTBL is terminated with a value of zero.

To set R2 to be used with the example, use the following instruction:

```
LI    R2,ADRTBL
```

The values passed to S$BIDT are used to build an Execute Task SVC block which S$BIDT then issues. S$BIDT also passes the synonyms of the calling task to the called task.

S$RTCA — Release Terminal Communications Area. This routine must be called by a command processor after it no longer needs access to the TCA. Usually, just before terminating.

Calling Sequence:	BLWP @S$RTCA
Registers Used:	R0 — Error code returned by S$RTCA

Example:

```
        .
        .
        .
*THIS ROUTINE IS FINISHED WITH THE TCA
BLWP @S$RTCA RELEASE TCA
        .
        .
        .
```

S$NEW — Initialize the System Data Base. This routine initializes a terminal communications area (TCA) for use by the various system routines according to the terminal state, mode, and ID. Command processors that do not call S$GTCA, but use other S$xxxx routines, must call S$NEW before using any of the other routines. Processors that do call S$GTCA need not call S$NEW, since S$GTCA does so.

Calling Sequence: BLWP @S$NEW

Registers Used: R0 — Error code returned by S$NEW

Example:

```
*THIS IS THE BEGINNING OF THE ROUTINE.
BEGIN BLWP @S$NEW
        .
        .
        .
        .
```

S$PARM — Get the I-th Parameter. This routine may be used by a command processor to get the parameters in the TCA which were passed to it by the command procedure through the PARMS parameter of a .BID, .QBID, or .DBID primitive. These parameters are text strings, delimited by commas, as shown under the .BID primitive description. Register Ra contains an integer that is the number of the parameter desired. Register Rb points to a buffer, into which the text string is to be copied. The first byte of the buffer must contain the length of the buffer. If the buffer is too short, an error code is returned in R0. Registers Ra and Rb are specified in the two bytes immediately following the call to S$PARM.

Calling Sequence:

```
BLWP @S$PARM
BYTE Ra, Rb
```

Registers Used:

R0 — Error code returned by S$PARM
Ra — Parameter number
Rb — Pointer to the buffer for the parameter text string

Example:

```
PARMI   BYTE    28          BUFFER FOR PARAMETER
        BSS     28
        .
        .
        .
        LI      R2,PARMI    R2 = POINTER TO BUFFER
        LI      R1,3        R1 = 3 (GET THE 3RD PARM)
        BLWP    @S$PARM
        BYTE    R1,R2       DEFINE 'A' AND 'B'
        .
        .
        .
```

S$SETS — Set Synonym Value. This routine is used to define or redefine a synonym in the terminal communications area. The synonym is a text string, pointed to by register Ra. The value to be assigned to the synonym is a text string pointed to by register Rb. If register "b" is zero, or points to a zero length string, the synonym is deleted from the TCA.

NOTE

In order to update the TCA with the new synonym value, S$PTCA must be called after the last call to S$SETS.

Calling Sequence:

```
BLWP @S$SETS
BYTE Ra, Rb
```

Registers Used:

R0 — Error code returned by S$SETS
Ra — Pointer to synonym name text string
Rb — Pointer to synonym value text string

Example:

```
        .
        .
        .
SYN04   BYTE    5           LENGTH OF SYN04 NAME
        TEXT    'SYN04'
VALUE1  BYTE    15          LENGTH OF VALUE
        TEXT    'DS02.CAT1.INPUT'
        .
        .
        .
        LI      R3,SYN04    R3 = POINTER TO SYN NAME
        LI      R7,VALUE1   R7 = POINTER TO SYN VALUE
SETS    BLWP    @S$SETS     DEFINE SYN
        BYTE    R3,R7       DEFINE 'A' AND 'B'
        .
        .
        .
```

S$MAPS — Map Synonym (Get Its Value). This routine will search the terminal communications area for the synonym name pointed to by Ra. If the synonym is found, and the buffer is large enough, its value is placed in the buffer pointed to by Rb, and the length of the value string is placed in the first byte of the buffer. The buffer is a text string buffer, with the first byte containing the length of the buffer minus one. If the buffer is too small, an error code will be returned in R0. If the synonym is not found in the TCA, a zero-length string is copied into the buffer. If the synonym name pointed to by Ra has a "." in it, the text preceding the "." will be replaced by its synonym value, if any exists, and the remainder of the synonym name will be copied into the value buffer without modification.

Calling Sequence:

```
BLWP @S$MAPS
BYTE Ra,Rb
```

Registers Used:

R0 — Error code returned by S$MAPS
Ra — Pointer to synonym name
Rb — Pointer to buffer for synonym value

Example:

```
                .
                .
                .
SYN05   BYTE    10              LENGTH OF SYN05
        TEXT    'SYN04.DATA'
VAL05   BYTE    50              LENGTH OF VALUE
        BSS     50
                .
                .
                .
        LI      2,SYN05         R2 = POINTER TO NAME
        LI      3,VAL05         R3 = POINTER TO BUFFER
MAPS    BLWP    @S$MAPS         GET SYN VALUE
        BYTE    R2,R3
```

S$SNCT — Search Name Correspondence Table. This routine searches the synonym table in the TCA for any synonym that is the immediate alphabetical predecessor or successor of the text string pointed to by register Ra. Finding the predecessor or successor depends on the value in R0.

If the desired synonym is found, the name is placed the buffer pointed to by Ra. If no synonym is found, a null string (zero length) is placed in the buffer pointed to by Ra. If Ra originally points to a zero length string, the alphabetically smallest synonym and its value are returned. If Rb is equal to zero, any synonym found is returned in the Ra buffer, but the corresponding value is not returned. If Rb is nonzero, the value of the synonym found is copied into the buffer pointed to by Rb. This routine is intended to be used to access synonyms in alphabetical order. Parameters entered by the PARMS parameter of a .BID, .QBID, or .DBID primitive are accessed using S$PARM (Get the Ith Parameter) routine.

NOTE

S$SNCT assumes that the buffer pointed to by Ra is 256 bytes long, and that the character count value in the first byte is a count of the string currently in the buffer. This is the only S$ routine that does not check buffer length before writing in the buffer.

Calling Sequence:

```
BLWP @S$SNCT
BYTE Ra,Rb
```

Registers Used:

R0 — If 0, the alphabetical successor is searched for; if −1, the alphabetical predecessor is searched for

Ra — Pointer to buffer containing original string; if a synonym name is found, it is placed here

Rb — Pointer to the buffer to receive the synonym value

Example:

```
               .
               .
               .
SYN       BYTE      3           LENGTH OF SYN BUF
NAME      TEXT      'SYN'
          BSS       252
VALUE     BYTE      255         LENGTH OF VALUE BUF
          BSS       255
               .
               .
               .
          LI        R0,0        R0 = GET SUCCESSOR
          LI        R3,SYN      R3 = NAME OF SYN
          LI        R4,VALUE    R4 = VALUE BUFFER
GETNXT    BLWP      @S$SNCT     GET NEXT SYN
          BYTE      R3,R4       DEFINE 'A' AND 'B'
          CLR       R1          R1 = 0
          MOVB      *R3,R1      CHECK FOR
          JEQ       OUT         END OF NCT
*PROCESS THE SYNONYM
               .
               .
               .
OUT
```

S$STAT — Get the Status of the Terminal. This routine returns the status of the terminal from which the calling routine (command processor) was activated. The information is returned in four bytes (two words), which have the following meaning:

Byte	Bit	Meaning
1	0	Reserved
	1–3	User privilege code, 0–7, as defined by the Assign User ID (AUI) command
	4–7	The current terminal mode, in hexadecimal, as follows: >00 = batch mode or background >01 = TTY mode >0F = VDT mode
2		Station ID
3		Reserved
4		The value of the CODE parameter from the most recently executed .BID, .QBID, or .DBID primitive.

The calling sequence and registers used are as follows:

Calling Sequence: BLWP @S$STAT

Registers Used:

R0 — Error code returned by S$STAT
R3 — Pointer to a 32-bit buffer

Example:

```
        EVEN
INFO    BSS     1           TERMINAL INFO
ID      BSS     1           STATION ID
        BSS     1           RESERVED
CODE    BSS     1           'CODE'
        LI      R3,INFO     R3 = POINTER TO BUFFER
        BLWP    @S$STAT     GET STATUS
```

S$SPLT — Split List into Components. This routine is used to remove the first element of a list, in the same manner as the .SPLIT primitive copies the first element (all text up to the first comma) of the list pointed to by R1 into the buffer pointed to by R2, putting the rest of the list into the buffer pointed to by R3. R1 and R3 may point to the same buffer.

Calling Sequence: BLWP @S$SPLT

Registers Used:

R0 — Error code returned by S$SPLT
R1 — Pointer to list text string
R2 — Pointer to buffer to receive first element of the list
R3 — Pointer to buffer to receive the rest of the list

Example:

```
        LIST     BYTE   32                          LENGTH OF LIST
                 TEXT   '(20,LIST ACCESS'
                 TEXT   'NAME,OUTPUT FILE)'
        FIRST    BYTE   20                          LENGTH OF 'FIRST' BUF
                 BSS    20
                  .
                  .
                  .
                 LI     R1,LIST                     R1 = LIST POINTER
                 LI     R2,FIRST                    R2 = FIRST POINTER
                 MOV    R1,R3                       R3 = REST POINTER = R1
                 BLWP   @S$SPLT
                  .
                  .
                  .
```

S$STOP — Return to the System Command Interpreter. This routine is used to terminate a command processor by returning control to SCI. It can be called at any point within the command processor. If S$CLOS was called earlier, with R1 = 0, the terminal local file will be displayed when control returns to SCI. If, upon calling S$STOP, R2 is not zero, the message text pointed to by R2 will be displayed after the terminal local file. This allows command processors to return error or completion messages. The value of R1 is converted to a four-digit hexadecimal integer and assigned as the value of synonym $$CC. The maximum message length for the routine S$STOP is 77 characters.

Calling Sequence: BLWP @S$STOP

Registers Used:
- R1 — Completion code. If zero, no error occurred.
- R2 — Pointer to a text string. If zero, no message is displayed.

The value in R1 is converted to hexadecimal ASCII, and assigned to the synonym $$CC if the processor was executed using .BID, or the synonym $$BC if the processor was executed using .QBID. If this processor is started by .QBID from a batch SCI, both $$CC and $$BC are assigned the value in R1.

Completion messages are passed to the foreground SCI via the TCA. If there is not available space in the TCA to hold the message, the message is not passed to the TCA and no error message is displayed to the user.

```
Example: ERRO     BYTE   18                      MESSAGE LENGTH
                  TEXT   'NORMAL TERMINATION'
                  MOV    @ERRCOD,R1              R1 = ERROR CODE
                  JEQ    ERRET                   JUMP TO ERROR RETURN
         RETNRM   LI     R2,ERRO                 R2 = MESSAGE
                  BLWP   @S$STOP
         ERRET    LI     R2,0                    DON'T RETURN MESSAGE
                  BLWP   @S$STOP                 RETURN TO SCI
                  END
```

6.3.8.3 Arithmetic Utility Subroutines. Four routines perform addition, multiplication, division, and subtraction using 32-bit signed integers as operands. Each routine also sets the status register bits 0 through 2, as do the normal assembly language arithmetic instructions.

S$IADD — Add 32-bit Integers. This routine adds two 32-bit integers (two's complement numbers), yielding a 32-bit integer. The two operands are pointed to by R1 and R2, and the result is placed in the 32-bit buffer pointed to by R3. Any of the registers may point to the same 32-bit area (add A to A, giving A). The status register arithmetic bits are set, as in the normal assembly language add instruction.

Calling Sequence: BLWP @S$IADD

Registers Used:

R0 — Error code returned by S$AIDD (−1 means overflow)
R1 — Pointer to 32-bit integer
R2 — Pointer to 32-bit integer
R3 — Pointer to 32-bit buffer for result

Example:

```
NUM1    DATA   1,0       FIRST NUMBER = 64K
NUM2    DATA   0,>FFFF   SECOND NUMBER = 64K-1
        LI     R1,NUM1   R1 = NUM1 POINTER
        LI     R2,NUM2   R2 = NUM2 POINTER
        MOV    R2,R3     R3 = RESULT POINTER
        BLWP   @S$IADD   ADD NUM1 + NUM2 GIVING NUM2
        .
        .
        .
```

S$ISUB — Subtract 32-bit Integers. This routine is used to subtract 32-bit integers and works like S$IADD. If R1 is zero, the negative of the number pointed to by R2 is calculated.

Calling Sequence: BLWP @S$ISUB

S$IMUL — Multiply 32-bit Integers. This routine multiplies the two 32-bit integers pointed to by R1 and R2 and places the result in the 32-bit buffer addressed by R3. Any of the registers may address the same 32-bit area. This routine works the same as S$IADD and S$ISUB.

Calling Sequence: BLWP @S$IMUL

S$IDIV — Divide 32-bit Integers. This routine divides the 32-bit integer pointed to by R1 by the 32-bit integer pointed to by R2 and places the quotient in the 32-bit buffer pointed to by R3 and the remainder in the 32-bit buffer addressed by R4. Any of the registers may address the same 32-bit area. If R3 = R4, only the quotient is stored. The status register arithmetic bits are set, as in S$IADD.

Calling Sequence: BLWP @S$DIV

Registers Used:

R0 — Error code returned by S$IDIV (divide by zero = −1)
R1 — Address of the dividend
R2 — Address of the divisor
R3 — Address of the quotient
R4 — Address of the remainder

Example:

```
NUMBER  DATA   8,0          NUMBER = 80000
        .
        .
        .
        LI     R1,NUMBER    R1 = NUMBER
        MOV    R1,R2        R2 = NUMBER
        MOV    R1,R3        R3 = NUMBER
        STWP   R4           R4,R5 = REMAINDER
        AI     R4,R4*2
        BLWP   @S$DIV       DIVIDE
*NUMBER = 1, R4 AND R5 = 0
```

6.3.8.4 Terminal Local File (TLF) Access Subroutines. The TLF for a terminal is a file of ASCII data in displayable format. The following routines are provided for opening and closing such a file, and for constructing and writing records to the file. The task that uses this set of routines must be bid via a .BID, .QBID, .DBID, or .TBID primitive or an S$BIDT subroutine call. The record is assumed to be no more than 134 characters. Data items are written at specific columns, and each line is terminated by a call to S$WEOL. Carriage control characters are edited into the text by the routines as required.

S$OPEN — Open File. The S$OPEN routine opens the terminal local file (TLF) or a user-specified file for write access. If R1 contains zero, the TLF is assumed. Any other value must represent the address of text containing the pathname of a file.

The TLF is used to to communicate informaton from a command processor to the user through short messages or listings. Each terminal has two terminal local files, one for foreground and one for background.

Calling Sequence: BLWP @S$OPEN

Registers Used:
R0 — Error code returned by S$OPEN
R1 — Zero or address of text containing the pathname of a file

Example:

```
CLR     R1          R1 SET FOR TLF
BLWP    @S$OPEN     OPEN TLF
        .
        .
        .
```

S$WRIT — Write to the Terminal Local File. This routine concatenates the text string addressed by R1 with the line currently being written to the TLF. If R2 is positive or zero, it specifies the column (0..133) in which the text should begin. The text string assumes the form described previously, with the first byte containing the length of the string. Additionally, if the byte value > 7F is encountered in the string, then the preceding character is repeated n times, where n is the value in the byte following the > 7F. The string should not contain device control characters, such as line feed, since these are supplied by S$WRIT as needed.

Calling Sequence: BLWP @S$WRIT

Registers Used:
R0 — Error code returned by S$WRIT
R1 — Address of text to be written
R2 — Column position at which text is to be placed

Example:

```
*ASSUME THAT S$WRIT HAS BEEN CALLED,
AND
*R2 POINTS TO THE END OF THE LAST WRITE
NEWTXT          BYTE    3               LENGTH OF NEW TEXT
                TEXT    '*'             THIS TEXT IS A
                BYTE    >7F             STRING OF ASTERISKS
REPEAT          BYTE    0               REPEATED
*THIS CODE WILL FILL THE 80 CHAR LINE WITH
ASTERISKS
LAB             LI      R3,80           CALCULATE NUMBER
                S       R2,R3           OF ASTERISKS NEEDED
                SWPB    R3
                MOVB    R3,@REPEAT
                LI      R1,NEWTXT       R1 = ADDRESS OF NEWTXT
                BLWP    @S$WRIT
                .
                .
                .
```

S$WEOL — Write End-of-Line to the Terminal Local File. This routine terminates the current line being written to the TLF and writes it to the file. If S$WRIT has not been called since S$OPEN or S$WEOL, a blank line is written.

Calling Sequence: BLWP @S$WEOL

Register Used: R0 — Error code returned by S$WEOL

S$CLOS — Close the Terminal Local File. This routine is used to terminate writing to the TLF. If the TLF is open, the S$CLOS routine should be called before the S$STOP routine. If R1 is zero, the file is displayed after the command completes (before the S$STOP message is displayed).

Calling Sequence: BLWP @S$CLOS

Registers Used: R0 — Error code returned by S$CLOS
R1 — If zero, the terminal local file will be displayed

Examples:

```
        .
        .
        .
        SETO    R1
BLWP    @S$CLOS         CLOSE,DON'T DISPLAY
        .
        .
        .
        CLR     R1
BLWP    @S$CLOS         DISPLAY THE TLF
```

6.4 SCI ENVIRONMENT AND BATCH STREAM OPERATION

Batch streams are command streams that control the operation of background tasks. A batch stream contains a stream of SCI commands and prompt responses that appear to the SCI task as if they were issued by a user at a terminal. However, since batch streams cannot interact with the terminal, you must supply field prompt responses within the batch stream.

Batch streams usually reside in a disk file. You execute them using the XB command.

The first line of a batch stream is a BATCH command and the last line is an EBATCH command. Before the first line is executed, the M$00 procedure is processed if it exists. After EBATCH, the M$01 procedure is executed if it exists. Then, the background SCI terminates.

The BATCH and EBATCH commands remove unnecessary SCI-generated synonyms from your synonym list for the background copy of the synonyms. To determine which synonyms are cleared in batch execution, execute the SF command to view the BATCH, EBATCH, and Q$SYN procedures.

NOTE

Any modifications that you make to the TI-supplied Q$SYN, BATCH, and EBATCH procedures may adversely affect the execution of TI installation batch streams. The contents of user-written M$00 and M$01 procedures may also affect batch execution.

In addition to the BATCH and EBATCH commands, batch streams can contain both SCI primitives and SCI commands. SCI primitives have already been discussed in detail. When using an SCI command in a batch stream, you must supply the following information:

- The command itself
- The keywords (prompting messages, and so on) associated with the command
- The parameter values (user responses) assigned to the keywords.

Use the following format for supplying this information:

< command> K1 = < value> ,K2 = < value> , . . .Kn = < value>

The command can be any SCI command. (All SCI commands except Text Editor and Debugger commands are fully documented in Volume II.) The keywords indicated by K1, K2, and so on, are the actual prompting messages that appear on the screen when you execute the command interactively; the value of that keyword is coded into the command instead of being supplied interactively.

NOTE

The keyword can be either the full prompting message, or an abbreviation of the prompting message that includes enough characters to make it recognizable as the intended prompt. A near equality algorithm compares the full keywords with their abbreviations. Often, only the first character of a keyword is required. This is shown in several of the examples used in this section. Refer also to the discussion of the near equality algorithm.

The BATCH command procedure bids an SCI task in background. It uses the .QBID primitive to bid the SCI task, so the terminal communications area synonyms are handled the same as with any other .QBID. That is, a copy is made of the foreground synonyms into background. These synonyms are then used by the batch stream just as a foreground SCI task would use foreground synonyms. The BATCH command procedure also executes the Q$SYN command procedure to release certain synonyms. If the batch stream you create needs synonyms normally released by Q$SYN, duplicate the Q$SYN command procedure in a separate command directory, and modify it not to release the needed synonyms.

The background SCI task executes the M$00 log-on procedure before processing commands from the batch stream, if the procedure is present in the command directory used. The M$01 procedure, if present, is executed when the background SCI terminates. Since these procedures may perform operations that you do not want performed upon entry and exit from a batch stream (such as deleting synonyms at log-off), you may want to prevent the rest of these procedures from being executed if called from background. You can do this by including the following code at the beginning of the M$00 and M$01 procedures:

```
! DO NOT EXECUTE REMAINDER OF M$01 IF CALLED BY BACKGROUND
! EXECUTION.

.IF @$$MO,EQ,"00"                  !TEST MODE SYNONYM.
.EXIT                              !EXIT PROC IF MODE IS BACKGROUND.
.ENDIF
```

When you issue a .QBID within a batch stream, .QBID functions like .BID when issued from a command running in foreground; that is, the batch stream suspends until the .QBID task completes. This means that synonyms are passed back from a batch stream to a parent batch stream.

Use the Text Editor to create a batch stream and store it in a disk file. The technique for batch streams is the same as for command procedures. However, a batch stream does not have to be in a particular directory.

You initiate a batch stream using the XB command. XB can be executed in response to the SCI prompt ([ ]), or as part of a command procedure.In either case, two prompts must be supplied to the XB command: Input Access Name and Listing Access Name. The input access name is the name of the batch stream file. The listing access name is either a printer unit ID or disk file pathname on which the batch stream operating report can be recorded.

In the following example, the batch stream called PRTPAYB prints a series of payroll reports. The batch stream assumes the reports were generated earlier, and now reside on disk in a directory called REPORTS. As the PRTPAYB batch stream executes, it produces a listing file, shown as an example in paragraph 6.5.3.

```
BATCH                          !PRTPAYB P/R BATCH
PF FILE=REPORTS.PAYROL1,  LD=LP01
PF FILE=REPORTS.PAYROL2,  ANSI=Y, LD=LP01
PF FILE=(REPORTS.PAYROL3,REPORTS.PAYROL4),  LD=LP01
PF FILE=REPORTS.PAYROL5,  LD=LP02
EBATCH TEXT=THE PAYROLL REPORT PRINTING IS COMPLETE
```

The following is an example of an SCI command procedure coded to call and execute the PRTPAYB batch stream. This calling command procedure is named PRTPAY. The command procedure assumes the batch stream resides in the PAYLIB directory, and the listing file that generates as the batch stream executes is stored in the file called PAYLIB.PRBTCH.

```
PRTPAY(PRINT PAYROLL REPORTS)
XB,
IAN=PAYLIB.PRTPAYB,
LAN=PAYLIB.PRBTCH
MSG TEXT=PAYROLL REPORT PRINTING BEGUN
```

Both the command procedure (PRTPAY) and the batch stream (PRTPAYB) are disk files, but they need not be in the same directory. The command procedure must be in an accessible command directory, but the batch stream can be in any directory, since the XB command allows entry of the full pathname.

The command procedure, PRTPAY calls the XB command, and passes to it the batch stream pathname and a listing access name. As soon as the XB command is activated, PRTPAY displays the message PAYROLL REPORT PRINTING BEGUN. The terminal then returns to foreground mode, activating SCI at that terminal.

The XB command procedure initiates the batch stream PAYLIB.PRTPAYB. In the preceding example, the input access prompt abbreviation is IAN, and the listing access prompt is LAN. Coding them into the procedure makes it unnecessary for the terminal user to know the pathnames of the files. The alternative method is to activate the PRTPAYB batch stream by entering XB in response to the SCI prompt, and supplying both the input file pathname and a listing file name.

The listing access name can be either a printer device ID or a disk pathname where a listing of the batch stream operation can be recorded. This listing is a detailed record of the execution of the batch stream, and it appears very similar to the batch stream itself. Any errors occurring during execution appear in this listing. A sample batch stream listing for the PRTPAYB batch stream is given in paragraph 6.5.3.

XB, BATCH, and EBATCH are the three command procedures required for any batch stream execution. The XB command can be initiated from the terminal or within another command procedure. The BATCH and EBATCH command procedures are called within the batch stream itself. All three of these command procedures can be found in the directory .SCI990.PROC0 on the system disk. You can view them using a SF command. The following list describes the processing steps performed when you execute a batch stream. These steps reference the preceding example, in which the batch stream is executed from a foreground command procedure called PRTPAY. Item 1 details the necessary user action, and the activity he observes. The remaining steps detail the processing activity.

1. When you enter PRTPAY in response to the SCI prompt ([ ]) and press the Return key, SCI displays the full procedure name, PRINT PAYROLL REPORTS. It then processes for a few seconds, and displays the message PAYROLL REPORT PRINTING BEGUN at the bottom of the screen. The message must be acknowledged by pressing the Return key to return to interactive SCI.

2. As soon as the PRTPAY procedure name appears on the screen, PRTPAY initiates execution of the XB command procedure. The IAN and LAN prompt responses are coded into the PRTPAY command procedure, so PRTPAY automatically passes these two pathnames to the XB command procedure.

3. SCI uses a *near equality* algorithm to identify the IAN and LAN field prompt abbreviations as INPUT ACCESS NAME and LISTING ACCESS NAME.

4. The XB command procedure assigns the values of IAN and LAN to synonyms. The input pathname .PAYLIB.PRTPAYB is assigned to the synonym XBI, and the listing pathname PAYLIB.PRBTCH is assigned to the synonym XBL. (The BATCH procedure references these synonyms to identify the batch stream pathname and listing device.)

5. The XB command procedure now issues the .QBID primitive which activates an SCI task in the associated terminal's background area.

6. The .QBID primitive also provides the background SCI task with a copy of the foreground synonyms resident in the terminal communications area of the associated terminal.

7. The XB command procedure (still executing in foreground), now clears the two synonyms used to pass the pathname values to the background SCI task. It clears the foreground copy of these synonyms to help keep the synonym table from overflowing, but they are still available to the background task.

8. The XB command procedure terminates after clearing the synonyms. The SCI menu, prompt ([ ]), and any messages appear.

9. Processing continues in the background. The background SCI task accesses the input access file PAYLIB.PRTPAYB, where the PRTPAYB batch stream resides.

10. The BATCH command procedure must be the first command in the batch stream. The BATCH command uses two standard SCI synonyms, $$UI and $$ST, to identify the user ID and station ID currently associated with the originating terminal. These IDs are printed on the batch stream listing. The listing access name and the listing device are identified through the XBI and XBL synonyms.

11. If the LS prompt in the BATCH command procedure is set to Y, the BATCH command also provides a list of the synonyms in the synonym table as part of the batch stream listing.

12. The BATCH command procedure verifies that it is executing in the background by checking the synonym $$MO. It terminates itself if it is not.

13. The BATCH command procedure clears several specific synonyms, and issues a Q$SYN. It then returns control to the batch stream or background SCI.

14. The next command in the example batch stream (PRTPAYB) is the PF command procedure, which is called to print the files. The values assigned to the field prompts in the PRTPAYB batch stream correspond to field prompts for the PF command. A pathname and listing device must be specified. Each PF command invoked in the batch stream requires its own printing parameters. This may be convenient if you need the reports printed at different terminals or locations, or if you were printing checks, and the check forms were already loaded onto a specific printer.

15. The EBATCH command procedure is executed next. The parameters of the EBATCH command include a text message. This message is assigned to the TEXT parameter, but does not display until the batch stream terminates.

16. The SDT command procedure now executes, placing the date and time on the listing report. SDT is invoked from within the EBATCH command.

17. EBATCH now terminates the batch stream with a .STOP primitive. The TEXT= assignment causes the background SCI to pass this message back to the foreground SCI.

18. The foreground SCI task is active at the terminal during the entire background processing, and the terminal user may be conducting other activities. The message from the batch stream appears on the terminal screen when the terminal user does one of the following things.

    - Terminates a foreground procedure
    - Presses the Return key in response to the SCI prompt
    - Initiates a WAIT SCI command
    - Presses the Command key

    Press the Return key to acknowledge the displayed message and then continue usual activities.

Within the batch stream itself, you can call any number of other SCI command procedures. This makes a batch stream a very efficient method of performing routine processing either in off hours, or when interactive terminals are occupied.

6.5 EXAMPLES

The following pages contain additional examples of SCI language component usage, such as prompts, keywords, and synonyms, by presenting example command procedures. A command processor example and a batch stream listing are also supplied.

6.5.1 Command Procedure Examples

The following command procedures represent examples only. They use actual SCI command procedure names to help you determine what function the command is performing.

```
AA(ADD ALIAS TO PATHNAME),
PATHNAME=ACNM("@$AA$P"),
ALIAS PATHNAME=ACNM
.OVLY   OVLY=>1B,LUNO=0,
        PARMS=(35,@&PATHNAME,@&ALIAS PATHNAME)
.SYN $AA$P="&PATHNAME"
```

In this example, a pathname is supplied as an initial value, by enclosing the string following the field prompt type in parentheses. Whatever value the synonym AAP has at the time is the initial value displayed. If you enter another pathname instead of accepting the initial value, that response is assigned to the synonym AAP at the end of the procedure so that it can appear as the initial value the next time AA is executed. The responses to the prompts PATHNAME and ALIAS PATHNAME can be synonyms. They are resolved when passed to the overlay.

```
BATCH (BEGIN BATCH EXECUTION),
USER ID                           = *STRING ("@$$UI"),
STATION ID                        = *STRING ("ST@$$ST"),
BATCH INPUT ACCESS NAME           = *STRING ("@$XB$I"),
BATCH LISTING ACCESS NAME         = *STRING ("@$XB$L"),
LS (LIST SYNONYMS) ?              = *YESNO (NO)
.IF @$$MO, NE, 0
  .EXIT
  .ENDIF
.SYN $XB$I = "", $XB$L = ""
.SYN $MR$ = "", $MRM$ = "", $MT$ = ""
.SYN $XD$D = "", $XD$F = "", $XD$S = ""
.SYN $CFK$L = "", $CFK$PN = "", $CFK$LRL = "", $CFK$KN = ""
.SYN $CFK$PRL = "", $CFK$KS = "", $CFK$IA = "", $CFK$M = ""
.SYN $CFK$SA = "", $CFK$MS = ""
.SYN $E$C = 0
Q$SYN
.IF "&LS", GE, "Y"
  LS L = ""
  .ENDIF
SDT
```

The BATCH command must also appear in any batch stream. The command procedure tests the variable $MO for the value 0, indicating batch mode. Defaults are provided for all the user prompts. If the synonym provided for the default is unassigned, no default appears on the screen.

```
XB (EXECUTE BATCH),
INPUT ACCESS NAME = ACNM,
LISTING ACCESS NAME = ACNM
.SYN $XB$I = "&INPUT ACCESS NAME"
.SYN $XB$L = "&LISTING ACCESS NAME"
.QBID TASK = > 20, PARMS = ("@&INPUT ACCESS NAME",
                            "@&LISTING ACCESS NAME")
.SYN $XB$I = "", $XB$L = ""
```

This XB command procedure in the preceding example is quite similar to the actual XB command procedure required to execute any batch stream. This command procedure uses the .QBID primitive to bid the task that actually executes the batch stream.

```
    DF(DELETE FILE) = 2,
    PATHNAME(S) = (ACNM)
    .SYN $DF = "(&PATHNAME(S))"
    .LOOP
    .SPLIT LIST="@$DF", FIRST=$DF1, REST=$DF
    .OVLY OVLY=>1B,PARMS=(8,@@$DF1)
    .WHILE "@$DF", NE, "$DF"
    .REPEAT
  .SYN $DF1=""
```

In this example, the parentheses around the prompt type shows that the response can be a list of pathnames. The loop within the procedure uses overlay code to delete one file at a time. It uses the .SPLIT primitive to determine when the last pathname has been processed.

6.5.2 Command Processor Example

Command processors are tasks bid from an SCI command procedure. They can access some interface subroutines supplied with the DX10 operating system. These routines are explained in preceding paragraphs, and perform functions such as opening the TCA so that the task can access the synonyms.

The command processor in the following example calls the interface subroutines S$GTCA, S$RTCA, S$STOP, and S$PARM. The command processor must contain a REF statement for each interface subroutine called in the code, although several routines can be referenced by one REF statement.

When S$STOP terminates the command processor, control returns to the command procedure at the statement following the .BID primitive that initiated the processor.

```
        IDT  'EXPRO'
*       THIS IS AN EXAMPLE COMMAND PROCESSOR TO BE CALLED
*       BY A PROCEDURE. IT GETS 2 PARAMETERS AND RETURNS
*       THEM AS A MESSAGE IN THE MESSAGE BUFFER. THE
*       MESSAGE IS THEN DISPLAYED WHEN THE RETURN TO THE
*       COMMAND INTERPRETER IS MADE.
                REF     S$GTCA,S$PARM,S$RTCA,S$STOP
                DATA    WS,PC,ERROR0
WS              BSS     32              PROCESSOR WORKSPACE
MSG             BYTE    255             TOTAL MESSAGE SIZE
                BSS     255             MESSAGE BUFFER
ERRO            BYTE    17
                TEXT    ':END ACTION TAKEN'
ERR1            BYTE    27
                TEXT    ':ERROR RETURNED FROM S$PARM'
ERR2            BYTE    27
                TEXT    ':ERROR RETURNED FROM S$GTCA'
```

```
PC          BLWP    @S$GTCA       GET THE TCA
            MOV     R0,R0         TERMINATE
            JNE     ERROR2             IF ERROR
            LI      R4,MSG        GO GET
            LI      R3,1               THE FIRST
            BLWP    @S$PARM            PARAMETER,WHICH IS
            BYTE    R3,R4              "&EXAMPLE NAME"
            MOV     R0,R0         TERMINATE
            JNE     ERROR1             IF ERROR
            MOVB    @MSG,R7       SET R7 = NUMBER OF
            SRL     R7,8               CHARACTERS READ IN
            A       R7,R4         R4 POINTS TO LAST CHARACTER
*                                      OF FIRST STRING
            NEG     R7            R7 = LENGTH OF
            AI      R7,255             REMAINING BUFFER
            MOVB    *R4,R6        R6 = LAST CHARACTER OF 1ST STRING
            SWPB    R7            R4 NOW POINTS TO THE
            MOVB    R7,*R4             REMAINING BUFFER
            LI      R3,2          GO GET THE SECOND
            BLWP    @S$PARM            PARAMETER,WHICH IS
            BYTE    R3,R4              "&NUMBER"
            MOV     R0,R0         TERMINATE
            JNE     ERROR1             IF ERROR
            AB      *R4,@MSG      @MSG CONTAINS LENGTH OF MESSAGE
            MOVB    R6,*R4        RESTORE LAST CHAR OF 1ST STRING
*         NORMAL NO ERROR RETURN
            LI      R2,MSG        RETURN MESSAGE
            CLR     R1
RETURN      BLWP    @S$RTCA       RELEASE TCA
            BLWP    @S$STOP       RETURN TO SCI
*         END ACTION
ERROR0      LI      R2,ERR0
            LI      R1,>8000
            JMP     RETURN
*         ERROR RETURN FROM S$PARM
ERROR1      LI      R2,ERR1
            LI      R1,>8000

            JMP     RETURN
*         ERROR RETURN FROM S$GTCA
ERROR2      LI      R2,ERR2
            LI      R1,>8000
            JMP     RETURN
            END
```

6.5.3 Batch Stream Listing

As batch streams execute, the execution results are sent to a file whose pathname you specify when you execute the XB command. The following example is a listing for the PRTPAYB sample batch stream presented in paragraph 6.4.

```
* SCI990   **   SCI990   **   SCI990   **   SCI990   **   SCI990   **BATCH


 <0001> BATCH
USER ID                         JF0012
STATION ID                      ST05
BATCH INPUT ACCESS NAME         .MYLIB.PRTPAYB
BATCH LISTING ACCESS NAME       .PRBTCH
LS (LIST SYNONYMS) ?            NO
15:55:32 WEDNESDAY, AUG 18, 1982.
 <0002> PF FILE =.PAYROL1, LD=LP01
FILE PATHNAME(S)                .PAYROL1
ANSI FORMAT?                    NO
LISTING DEVICE                  LP01
DELETE AFTER PRINTING?          NO
NUMBER OF LINES/PAGE            ** NULL **
 USER   DEVICE  STATUS   FILE NAME
  JF0012  LP01   ACTIVE   .PAYROL1
 <0003> PF FILE =.PAYROL2, ANSI=Y, LD=LP01
FILE PATHNAME(S)                .PAYROL2
ANSI FORMAT?                    Y
LISTING DEVICE                  LP01
DELETE AFTER PRINTING?          NO
NUMBER OF LINES/PAGE            ** NULL **
 USER   DEVICE  STATUS   FILE NAME
  JF0012  LP01   ACTIVE   .PAYROL2
 <0004> PF FILE =(.PAYROL3,.PAYROL4), LD=LP01
FILE PATHNAME(S)                .PAYROL3,.PAYROL4
ANSI FORMAT?                    NO
LISTING DEVICE                  LP01
DELETE AFTER PRINTING?          NO
NUMBER OF LINES/PAGE            ** NULL **
 USER   DEVICE  STATUS   FILE NAME
  JF0012  LP01   ACTIVE   .PAYROL3
  JF0012  LP01   WAITING  .PAYROL4
 <0005> PF FILE =.PAYROL5,LD=LP02
FILE PATHNAME(S)                .PAYROL5
ANSI FORMAT?                    NO
LISTING DEVICE                  LP02
DELETE AFTER PRINTING           NO
NUMBER OF LINES/PAGE            ** NULL **
 USER   DEVICE  STATUS   FILE NAME
  JF0012  LP02   ACTIVE   .PAYROL5
 <0006> EBATCH TEXT=THE PAYROLL REPORT PRINTING IS COMPLETE
16:10:22 WEDNESDAY, AUG 18, 1982.
```

6.6 ERROR MESSAGES

Some of the errors likely to occur as a result of preparing SCI procedures are discussed in the following paragraphs. A complete listing of SCI error messages can be found in the *DX10 Error Reporting and Recovery Manual* (Volume VI).

6.6.1 Unknown Volume Name

This error indicates an attempt to open a file with an incorrect file name. A common reason for this error in a procedure, forgetting to code the ampersand (&) in front of a prompt name, is shown in the following example:

```
.SHOW FILE PATHNAME
```

This example is incorrectly interpreted by SCI as a request to show the file named FILE PATHNAME. An ampersand must be coded in front of prompt names to identify them to SCI. The correct format is as follows:

```
.SHOW &FILE PATHNAME
```

6.6.2 9001 — Invalid Access Name Syntax

This error can result from a failure to encode the at sign (@) in front of a synonym name as in the following example:

```
.SHOW $SF$P
```

This primitive results in the 9001 error message. The synonym must be coded as @SFP.

6.6.3 9003 — Invalid Keyword Syntax

A prompt has been improperly defined. Examine the prompts in the procedure for syntax errors.

6.6.4 9005 — Invalid Command Name Syntax

This error usually occurs because the comma required after the command name is missing, as shown in the following examples:

```
EXP(EXAMPLE PROC) FILE PATHNAME = ACNM

EXP(EXAMPLE PROC)
FILE PATHNAME = ACNM
```

A comma is required after the command name:

```
EXP(EXAMPLE PROC), FILE PATHNAME = ACNM

EXP(EXAMPLE PROC),
FILE PATHNAME = ACNM
```

6.6.5 9006 — Invalid Relation Name

This error is caused by using an invalid relational operator in a command procedure. Refer to the .IF, .ELSE, and .ENDIF primitives.

6.6.6 9007 — Invalid Type Specification

This error is caused by a prompt type other than ACNM, STRING, NAME, INT, or YESNO as in the following example:

```
FILE PATHNAME = ACNN
```

The misspelled prompt type (ACNN) in this line causes the 9007 error.

6.6.7 900A — Spurious Characters at End

This error can occur as a result of improper use of double quotes as in the following example:

```
MSG TEXT = "ENTER "RETURN" TO CONTINUE"
```

Quoted strings require two double quotes within the string. The correct form for this command is as follows:

```
MSG TEXT = "ENTER ""RETURN"" TO CONTINUE"
```

6.6.8 900E — Unknown Command Name

Any procedure line in which SCI cannot identify a primitive or prompt name will result in this error. For example,

```
EXP(EXAMPLE PROC)=0
FILE PATHNAME = ACNM
```

This example is incorrect because there is no comma after the procedure name line.

6.6.9 900F — Unknown Keyword

A prompt has been referenced in the procedure which is not defined in a prompt list and does not fit any of the standard rules of abbreviation. Verify the prompt names used in the procedure.

6.6.10 9011 — Required Argument Not Present

When an SCI procedure is called from another procedure, the calling procedure may be required to pass the prompt values. If the calling procedure is in foreground mode, it can call another procedure and pass no prompt values. In this case you are required to enter the prompt values in the normal fashion. If the calling foreground procedure wishes to pass any prompt values automatically it must pass *all* of the required prompt values; otherwise, the 9011 error will occur.

Required prompts are prompts that are not given initial values and are not declared optional by placing an asterisk in front of the field type modifier. In the following example, only the first prompt is required.

```
FILE PATHNAME = ACNM,
OUTPUT FILE = *ACNM,
DELETE FILE = YESNO(NO),
PRINT FILE = ACNM(@$SF$P)
```

OUTPUT FILE is an optional response. DELETE FILE is set to an initial value. PRINT FILE shows an initial value if SFP has an assigned value; otherwise, the synonym will resolve to a null string because it begins with a $ sign. In the background mode, a procedure is required to pass all required values to any other called procedures. This is necessary because background procedures cannot access the terminal.

6.6.11 9019 — Invalid Keyword Value

The value of a prompt is of a form not allowed for that prompt type. An INT type prompt cannot contain alphabetical values greater than F.

6.6.12 FF02 — PROC Library Error

This error occurs when the procedure name is not the same as its file name.

EXP(EXAMPLE PROC) = 0,

If the file name for the example is .S$PROC.SF, an FF02 error occurs when the procedure is invoked.

6.6.13 FF0B — Keyword Table Overflow

This error indicates that the workspace that SCI uses to store prompt values has filled up. This error can be caused by a number of things. Recursive calls of procedures, or failure to release temporary files at the end of the procedure are common problems. The keyword table occupies the same memory space as the synonym table.

When a procedure calls another procedure, the two procedures use the same workspace. Therefore, recursive calls of procedures by other procedures can cause the keyword table to overflow.

Within a single procedure, enough information could possibly be generated to overflow the keyword table. While this is highly unlikely, several things could lead to the condition. First, very long lists of file names entered as responses can cause overflow. Second, a large number of very long prompt names may cause the condition because the keyword table stores both the prompt name and its associated data.

7

Using Supervisor Calls (SVCs)

7.1 INTRODUCTION

User tasks interface with the DX10 operating system by issuing supervisor calls (SVCs), which request that the operating system perform a specific function. SVCs are provided within DX10 to perform device and file I/O, task control, service functions, memory control, and file utilities.

SVCs are explicitly used in assembly language tasks, whereas high-level language statements are processed by the appropriate interpreter or compiler and are translated to the particular SVC required to perform the requested operation. If a statement to access a particular SVC is not available in the high level language, you can write an assembly language subroutine to access the SVC. Some high level languages support calls to SVCs through special coding practices. Refer to the applicable language programmer's guide for further information on this technique.

To use SVCs, you do not need to be an accomplished assembly language programmer; however, you do need to be familiar with general concepts and practices of assembly language programming, as presented in earlier sections of this manual.

The following paragraphs define a supervisor call and the format for writing a supervisor call in assembly language.

7.2 SUPERVISOR CALL DEFINITION

SVCs are predefined routines in the operating system that perform functions generally required by assembly language programs, such as opening files, assigning LUNOs, and so on. You can also define your own SVCs for special purposes, if they are not supplied on DX10. Refer to the *DX10 Systems Programming Guide* (Volume V) for details.

The three categories of SVCs are as follows:

- Program support SVCs
- Device I/O SVCs
- File I/O SVCs

Program support SVCs control memory use, control program execution, and perform some miscellaneous functions such as data conversion among binary, decimal, and hexadecimal representations. Program support SVCs are discussed individually in Section 8. Appendix F lists all DX10 SVCs and the SVC opcodes for easy reference.

Device I/O SVCs control both device-dependent and device-independent I/O, by allowing you to code the call blocks differently, depending on the function desired. Device I/O SVCs control assigning and releasing LUNOs, and all read, write, and special functions for devices.

File I/O SVCs control I/O to files by controlling LUNO assignment and release, and all read and write operations to the file. Appendix E contains a call block coding chart showing both file and device I/O opcodes.

In an assembly language program, you must access code an SVC call block in order to access SVCs. The call block must contain the SVC code and any parameters required to complete the requested function. Then you must issue the XOP assembly language instruction using 15 as the XOP number, and the address of the coded SVC call block as the effective address of the XOP instruction. The XOP instruction initiates an extended operation, which causes execution control to be passed to the operating system. (For more detailed information on the XOP instruction, see the *990/99000 Assembly Language Reference Manual.*)

The supervisor call block referenced in the XOP instruction contains the necessary parameters for the requested operation. The first byte of the call block always contains the SVC code, which specifies the particular service routine to perform the needed operations. Figure 7-1 illustrates the processing steps occurring when you issue an XOP instruction for a supervisor call.

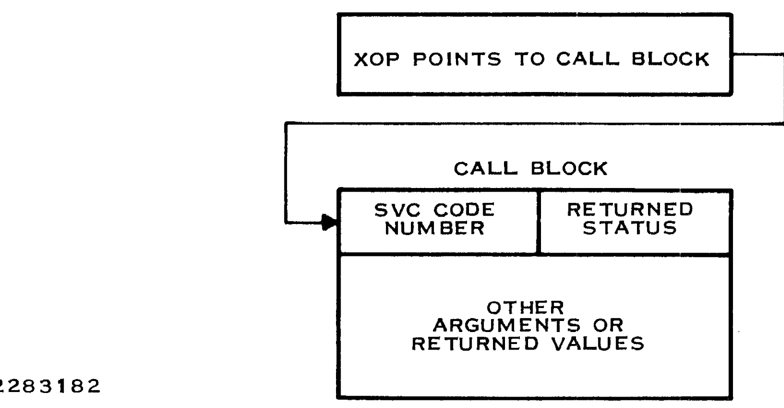


Figure 7-1. XOP Call Processing

7.2.1 Coding a Supervisor Call

Within the assembly language program, supervisor calls can be issued in either of two ways:

- Use the XOP instruction, with the extended operation level defined as 15 and the effective address being that of the supervisor call block.
- Use the DXOP instruction to define a new extended operation. DXOP allows you to assign a symbol that can be used in the operator field of the XOP instruction.

To use the XOP instruction, you must specify the address of the SVC call block as an argument. The following is an example of this coding technique using a supervisor call block that has been labeled BLOCK:

```
        XOP  @BLOCK,15 PERFORM EXTENDED OPERATION 15
    *                  USING THE SUPERVISOR CALL BLOCK AT
    *                  THE ADDRESS DEFINED BY THE SYMBOL
    *                  BLOCK
```

The second method of coding supervisor calls involves using the DXOP instruction to assign a symbol to the operator field of XOP instructions. Before coding a supervisor call, you can define the XOP level 15 to be equivalent to some symbol. That symbol can then be used instead of the XOP instruction when coding SVCs. To illustrate, all the examples showing how to code various SVCs in this document assume that the Extended Operation 15 has been defined by the following directive to be the symbol SVC, as follows:

```
    DXOP  SVC,15
```

This allows the symbol SVC to replace the XOP instruction in the operator field of the assembly language program. The following example shows an SVC coded after the DXOP instruction is executed:

```
    SVC   @BLOCK
```

The preceding instruction specifies that extended operation 15 be performed using the call block at the location defined by the symbol BLOCK.

7.2.2 Defining Supervisor Call Blocks

Supervisor call blocks are defined within the user task by the use of DATA, BYTE, and TEXT directives. Note that the DATA directive is used to define word entries (16 bits); the BYTE directive is used to define byte entries (8 bits). Bits are numbered from left to right. The TEXT directive is used to define character string entries. Since some of the call blocks must be aligned on a word boundary, either the DATA directive must be used, forcing alignment on a word boundary, or the definitions for the block of data must be preceded by an EVEN directive, which also forces the block to begin on a word boundary. Beginning all supervisor call blocks on a word boundary is a good coding practice.

The number of parameters required within the call block varies for each supervisor call. However, the first byte (byte 0) of the call block must contain the SVC opcode for the service required. The second byte (byte 1) of the call block is generally used by the system to return a status or error code. This enables you to test the error code and process any error that occurs by including the necessary statements in your program. Using the error code byte is discussed in more detail in Paragraph 7.2.3.

CAUTION

DX10 uses XOP R15,15 (alternatively SVC R15) to implement breakpoints. Do not use workspace register 15 as a supervisor call block.

When coding a supervisor call block, label entries used to store data returned by the system or entries containing variable data. This facilitates access of the data by the user task. The label associated with the first byte of the call block is used by the XOP and DXOP instructions as the effective operand to reference the block.

The following assembly language programming example performs one I/O cycle. The program assigns a LUNO, opens it, writes to it, closes it and finally releases it, using SVCs. For this discussion, note the SVC call blocks and the commands required to access them. This example also provides example usage of REF statements for the interface subroutines explained in Section 9.

```
TASK1         SDSMAC 947075 *E    08:37:34  TUESDAY, AUG 02, 19XX
                                                                   PAGE 0001
 0001                           IDT '´TASK1'
 0002                    *
 0003                    *TASK HEADER
 0004                    *
 0005                           REF  PROC1
 0006 0000 0078'                DATA TWS,PROC1,0         TRANSFER VECTOR
      0002 0000
      0004 0000
 0007                           DEF  ALUNO,ALNUM,OPEN,OLNUM
 0008                           DEF  WRITE1,W1LNUM,WRITE2,W2LNUM
 0009                           DEF  CLOSE,CLNUM,RELEAS,RLNUM
 0010                    *
 0011                    *ASSIGN LUNO BLOCK
 0012                    *
 0013 0006                      EVEN
 0014 0006    00         ALUNO  BYTE 0,0                 IO,STATUS
      0007    00
 0015 0008    91                BYTE >91                 ALUNO OPCODE
 0016 0009    00         ALNUM  BYTE >00                 LUNO NUMBER
 0017 000A 0000                 DATA 0,0,0,0,0,0         NOT USED
      000C 0000
      000E 0000
      0010 0000
      0012 0000
      0014 0000
 0018 0016    00         ALUFLG BYTE >04,0               UTILITY FLAGS-GENERATE LUNO
      0017    00
 0019 0018 0000                 DATA 0,0,DEVNAM,0,0,0,0,0,0
      001A 0000
      001C 00B5
      001E 0000
      0020 0000
      0022 0000
      0024 0000
      0026 0000
      0028 0000
 0020                    *
 0021                    *OPEN PRB
 0022                    *
 0023 002A                      EVEN
 0024 002A    00         OPEN   BYTE 0,0                 IO,STATUS
      002B    00
 0025 002C    00                BYTE 0                   OPEN
 0026 002D    00         OLNUM  BYTE 0                   LUNO, TO BE FILLED IN
 0027 002E    00                BYTE 0,0                 SYSTEM, USER FLAGS
      002F    00
 0028 0030    00                BYTE 0,0                 DEVICE TYPE
      0031    00
 0029 0032 0000                 DATA 0,0,0
      0034 0000
      0036 0000
```

```
0030                  *
0031                  *WRITE ONE PRB
0032                  *
0033 0038                    EVEN
0034 0038      00    WRITE1 BYTE 0,0                IO,STATUS
     0039      00
0035 003A      0B            BYTE >B                 WRITE ASCII
0036 003B      00    W1LNUM BYTE 0                   LUNO,TO BE FILLED IN
0037 003C      00            BYTE 0,0                SYSTEM,USER FLAGS

TASK1          SDSMAC 947075 *E   08:37:34  TUESDAY, AUG 02, 19XX
                                                                     PAGE 0002
     003D      00
0038 003E 0098'              DATA TEXT1              BUFFER ADDRESS
0039 0040 0010               DATA TEXT1L             CHAR COUNT
0040 0042 0010               DATA TEXT1L             CHAR COUNT
0041 0044 0000               DATA 0
0042                  *
0043                  *WRITE TWO PRB
0044                  *
0045 0046                    EVEN
0046 0046      00    WRITE2 BYTE 0,0                 IO,STATUS
     0047      00
0047 0048      0B            BYTE >B                 WRITE ASCII
0048 0049      00    W2LNUM BYTE 0                   LUNO, TO BE FILLED IN
0049 004A      00            BYTE 0,0                SYSTEM,USER FLAGS
     004B      00
0050 004C 00A8               DATA TEXT2              BUFFER ADDRESS
0051 004E 000D               DATA TEXT2L             CHAR COUNT
0052 0050 000D               DATA TEXT2L             CHAR COUNT
0053 0052 0000               DATA 0
0054                  *
0055                  *CLOSE PRB
0056                  *
0057 0054                    EVEN
0058 0054      00    CLOSE   BYTE 0,0                IO,STATUS
     0055      00
0059 0056      01            BYTE >1                 CLOSE
0060 0057      00    CLNUM   BYTE 0                  LUNO, TO BE FILLED IN
0061 0058      00            BYTE 0,0                SYSTEM,USER FLAGS
     0059      00
0062 005A 0000               DATA 0,0,0,0            NOT USED
     005C 0000
     005E 0000
     0060 0000
0063                  *
0064                  *RELEASE LUNO
0065                  *
0066 0062                    EVEN
0067 0062      00    RELEAS  BYTE 0,0                IO,STATUS
     0063      00
0068 0064      93            BYTE >93                RELEASE LUNO
0069 0065      00    RLNUM   BYTE 0                  LUNO,TO BE FILLED IN
0070 0066 0000               DATA 0,0,0,0,0,0,0,0,0  NOT USED
0068 0000
     006A 0000
     006C 0000
     006E 0000
     0070 0000
     0072 0000
     0074 0000
     0076 0000
```

```
 0071                *
 0072                *TASK 1 DATA
 0073                *
 0074 0078           TWS     BSS   16*2
 0075 0098    49     TEXT1   TEXT  'I AM TASK ONE.55'
      0099    20
      009A    41
      009B    4D
      009C    20
      009D    54

TASK1           SDSMAC 947075 *E    08:37:34   TUESDAY, AUG 02, 19XX
                                                                        PAGE 0003
      009E    41
      009F    53
      00A0    4B
      00A1    20
      00A2    4F
      00A3    4E
      00A4    45
      00A5    2E
      00A6    35
      00A7    35
 0076       000D     TEXT1L EQU   $-TEXT1
 0077 00A8    4E     TEXT2  TEXT  'NOW I AM DONE'.
      00A9    4F
      00AA    57
      00AB    20
      00AC    49
      00AD    20
      00AE    41
      00AF    4D
      00B0    20
      00B1    44
      00B2    4F
      00B3    4E
      00B4    45
 0078       000D     TEXT2L EQU   $-TEXT2
 0079 00B5    04     DEVNAM BYTE  >4
 0080 00B6    4C            TEXT  'LP01'
      00B7    50
      00B8    30
      00B9    31
 0081                       END
NO ERRORS

PROC1           SDSMAC 947075 *E     13:13:52 MONDAY, AUG 01, 19XX
                                                                        PAGE 0001
 0001                *
 0002                *PROCEDURE SEGMENT 1
 0003                *
 0004                       IDT 'PROC1'
 0005                       DEF PROC1
 0006                       REF PROC2
 0007                       REF ALUNO,ALNUM,OPEN,OLNUM
 0008                       REF WRITE1,W1LNUM,WRITE2,W2LNUM
 0009                       REF CLOSE,CLNUM,RELEAS,RLNUM
 0010                *
 0011                *PROCEDURE ONE
 0012                *
 0013       0000'    PROC1  EQU $
 0014 0000 2FE0             XOP  @ALUNO,15          ACCESS LUNO NUMBER
```

```
0015 0004 D020          MOVB @ALUNO,R0          MOVE LUNO NUMBER TO PRB
     0006 0000
0016 0008 D800          MOVB R0,@OLNUM
     000A 0000
0017 000C D800          MOVB R0,@W1LNUM
     000E 0000
0018 0010 D800          MOVB R0,@W2LNUM
     0012 0000
0019 0014 D800          MOVB R0,@CLNUM
     0016 0000
0020 0018 D800          MOVB R0,@RLNUM
     001A 0000
0021 001C 2FE0          XOP  @OPEN,15           OPEN LUNO
     001E 0000
0022 0020 2FE0          XOP  @WRITE1,15         WRITE TEXT 1
     0022 0000
0023 0024 0460          B    @PROC2             GO TO PROC 2
     0026 0000
0024                    END
NO ERRORS,     NO WARNINGS

PROC2          SDSMAC 947075 *E   08:43:22  TUESDAY, AUG 02, 19XX
                                                                PAGE 0001
0001                 *
0002                 *PROCEDURE SEGMENT 2
0003                 *
0004                        IDT   'PROC2'
0005                        DEF   PROC2
0006                        REF   WRITE2,CLOSE,RELEAS
0007                 *
0008                 *PROCEDURE TWO
0009                 *
0010          0000'  PROC2  EQU   $
0011 0000 2FE0              XOP   @WRITE,15    WRITE TEST 2
     0002 0000
0012 0004 2FE0              XOP   @CLOSE,15    CLOSE LUNO
     0006 0000
0013                 *      XOP   @RELEAS,15   RELEASE LUNO**DUMMY IF NOT FILE*
0014 0008 2FE0              XOP   @EOT,15      END OF TASK
     000A 000C'
0015                 *
0016                 *CONSTANT DATA
0017                 *
0018 000C    04      EOT    BYTE 4,0           END OF TASK
     000D    00
0019                        END
NO ERRORS
```

7.2.3 Returning the Error Code

Usually, byte one of the call block is used as the error code byte. If an error occurs during the execution of the requested SVC, the system returns an error code in byte one. It is the responsibility of the calling task to check this error byte to determine whether the requested operation has been completed normally. If the byte contains a nonzero value, an error has occurred and the calling task must provide the code to handle the error in the proper manner. Error codes are documented in the *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI).

It is often useful to define error code bytes within call blocks by a separate BYTE directive and labeling it with a descriptive label such as ERRC. This allows the calling task to easily access the byte to determine whether an error has occurred. The following series of statements could be used within the calling task to test the error code and branch to the correct routine if an error has occurred.

```
      MOVB    @ERRC,@ERRC                COMPARE THE ERROR CODE TO ZERO
      JNE     ERRTN                      IF NOT ZERO, GO TO ERROR ROUTINE
        .                                IF ZERO, CONTINUE NORMAL PROCESSING
        .
        .
```

8

Program Support Calls

8.1 GENERAL

Program support calls are SVCs used in application programs to request the services of DX10 for operations not directly related to I/O operations. Such services include placing tasks in execution, terminating tasks, synchronizing intertask communication, identifying tasks, and task end action. The program support SVCs fall into the following categories:

- Program control SVCs
- Memory control SVCs
- Intertask communications SVCs
- Data conversion SVCs
- System information SVCs

This section discusses the SVCs in each group. Table 8-1 shows program support SVCs by function. Appendix F lists all SVCs.

Usually, SVCs are accessible through assembly language programs and subroutines. The high-level languages use statements in their respective repertoires to achieve the same results, where applicable, as the supervisor calls. If no statement is provided within the high-level language that accesses the desired supervisor call, you can write an assembly language module that issues the supervisor call, and is callable from the high-level language program. You must link these assembly language routines to the high-level language object module using the Link Editor.

Some high-level languages can support SVC calls directly, provided the call is coded in a special way. The methods applicaable to each language are documented in the individual language programmer's guides.

The descriptions of individual SVCs presented in this section assume the following:

- The DXOP instruction has previously defined XOP 15 to be the mnemonic SVC. In all examples showing the assembly language code required to initiate an SVC, the mnemonic SVC is used. The DXOP directive initiates the call operation, using the SVC mnemonic and the value 15 as arguments. The DXOP directive is discussed in the preceding section.

- The call block examples use the label ERRC for the error code byte. The error code byte is usually byte one of the call block.

Table 8-1. Program Support Supervisor Calls

Call	Hexadecimal SVC Code
Program Control Calls	
Time Delay	02
End of Task	04
Unconditional Wait	06
Activate Suspended Task	07
Do Not Suspend	09
Activate Time Delay Task	0E
Change Priority	11
Load an Overlay	14
End of Program	16
Get Parameters	17
Schedule Bid Task	1F
Execute Task	2B
Self Identification	2E
End Action Status	2F
Map Program Name to ID	31
Poll Status of Task in Terminal Task Set	35
Reset End Action	3E
Memory Control Calls	
Get Common Data Address	10
Get Memory	12
Release Memory	13
Return Common Data Address	1B
Intertask Communications Calls	
Putdata	1C
Getdata	1D
Data Conversion Calls	
Convert Binary to Decimal	0A
Convert Decimal to Binary	0B
Convert Binary to Hexadecimal	0C
Convert Hexadecimal to Binary	0D
System Information Calls	
Date and Time Support	03
System Log	21
Retrieve System Information	3F

8.2 PROGRAM CONTROL SVCs

Program Control SVCs are SVCs that directly affect a task. They can execute a task, suspend a task, or terminate a task. The following paragraphs define each of the Program Control SVCs and give the format of the SVC call block, the SVC opcode, and the errors returned by the system. In addition, an example of each Program Control SVC is supplied.

8.2.1 >02 — Time Delay SVC

The Time Delay SVC is used to suspend the calling task for a specified minimum number of full 50-millisecond clock periods. Specifying 0 implies a delay from 0 to 50 ms; specifying 1 implies a delay from 50 ms to 100 ms, and so on. A task in time delay may be reactivated before the end of the specified time interval by another task issuing an Activate Time Delay Task SVC (opcode >0E). The following is an example of the block required to delay a task for 10 seconds:

```
SCBF   DATA   >0200    CALL CODE AND SET BYTE ONE TO ZERO
INTV   DATA    200     DELAY FOR 200 TIME INTERVALS
```

Within the procedure, the call to perform the Time Delay operation using the preceding call block is written as follows:

```
SVC            @SCBF
```

8.2.2 >04 — End of Task SVC

The End of Task SVC call is used to terminate execution of the calling task. The call block consists of one byte, which contains the call code, and need not be aligned on a word boundary. An example of the call block is as follows:

```
SCBE           BYTE 4      END OF TASK CALL OPCODE
```

The function is called as follows:

```
SVC            @SCBE
```

The End of Task SVC I performs such functions as releasing the local logical device tables, releasing task memory and the task status block for disk-resident tasks, reinitializing the task status block for memory resident tasks, and clearing of any outstanding breakpoints.

8.2.3 >06 — Unconditional Wait SVC

The Unconditional Wait supervisor call suspends the calling task indefinitely, or until another task issues an Activate Suspended Task SVC (opcode >07).

If an Activate Suspended Task SVC is issued to a task that is not suspended, then an Unconditional Wait SVC subsequently encountered will have no effect. That is, the task is reactivated immediately. If two Activate Suspended Task SVCs are issued to an active task, only the first Unconditional Wait SVC encountered has no effect. Therefore, only one Activate Suspended Task SVC can be "remembered" by the system.

The call block for the Unconditional Wait call consists of one byte, which contains the call code and does not need to be aligned on a word boundary. The following is an example of the Unconditional Wait call block:

```
SCBH     BYTE     6          SUSPEND CALLING TASK
```

The Unconditional Wait service is called from the procedure section of the task by the following statement:

```
SVC      @SCBH
```

8.2.4 >07 — Activate Suspended Task SVC

The Activate Suspended Task SVC reactivates a task that has placed itself in a suspended state using the Unconditional Wait SVC. You must specify the suspended task's runtime ID when you issue an Activate Suspended Task SVC.

The system remembers one (and only one) Activate Suspended Task SVC call for each task activated. If task A is not suspended, but task B issues an Activate Suspended Task SVC against task A, then task A will reactivate immediately after it issues an Unconditional Wait (opcode >06) SVC. However, if task B issues more than one Activate Suspended Task SVC against task A before task A suspends, only the first reactivation SVC is remembered, and task A immediately reactivates only once. Figure 8-1 illustrates this feature.

This provides a means of synchronizing tasks by allowing one task to wait for another task to complete an operation and then become reactivated. This is illustrated in the preceding example. If Task A requires that Task B complete some operation before it can continue, Task A issues an Unconditional Wait for itself, and when Task B completes the function, it issues an Activate Suspended Task call for Task A to reactivate it. If Task B completes its function and issues an Activate Suspended Task call that references Task A before Task A has suspended itself, no action is taken until Task A issues an Unconditional Wait, at which point it is immediately reactivated.

NOTE

In general, the Activate Suspended Task call is only effective for tasks that have been self-suspended, (that is, suspended using SVC opcode >06)

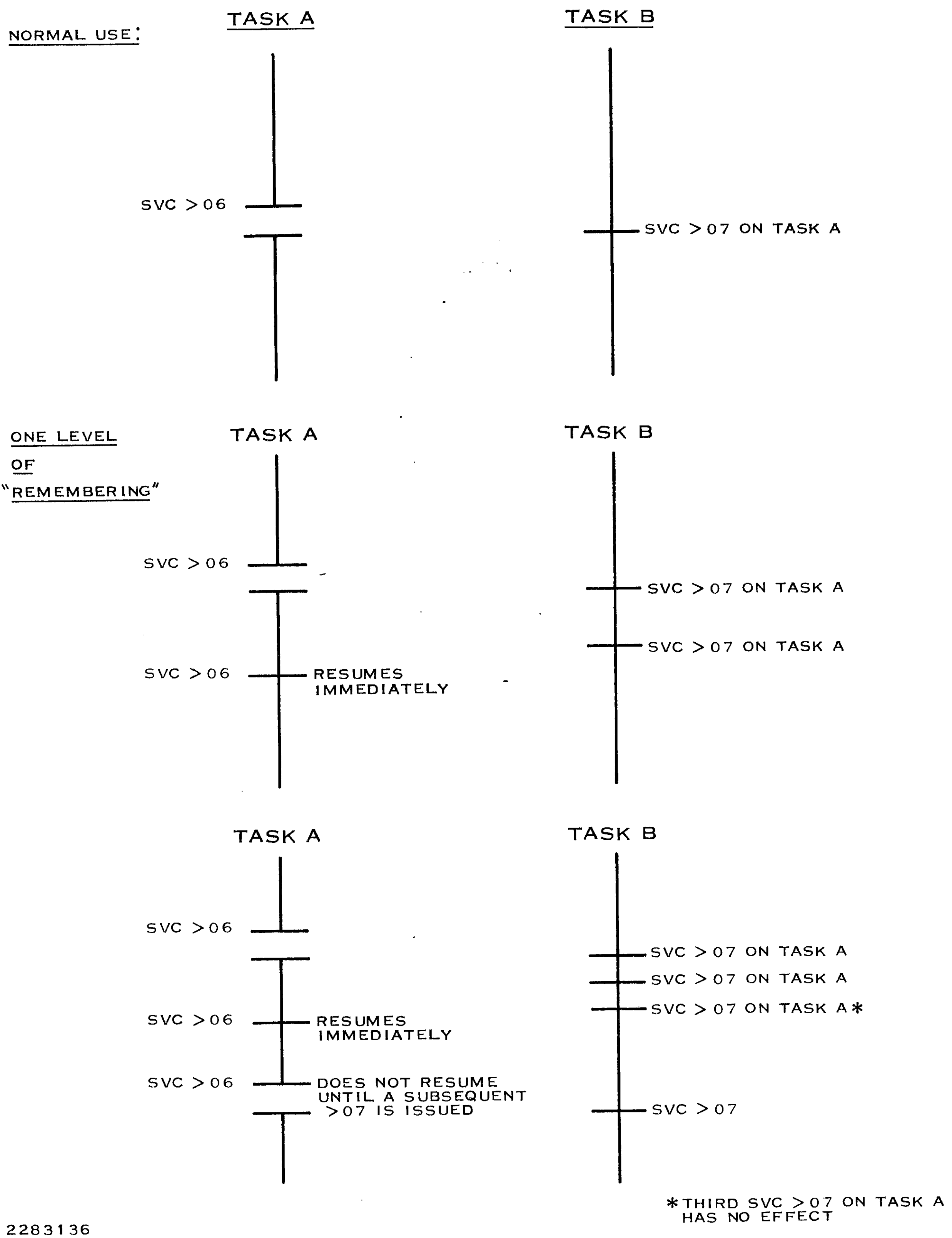


Figure 8-1. Task Synchronization

The SVC call block for the Activate Suspended Task supervisor call consists of three bytes and does not need to be aligned on a word boundary. The bytes have the following meanings:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >07.
1		The system returns the task state in this byte. If the task cannot be found, this byte contains the error code >FF. Refer to Appendix C for a list of possible task state codes.
2		The hexadecimal identifier of the task to be activated. Note that this is the run-time identifier, which usually differs from the installed identifier of the task.

This implies that if several tasks are to cooperate and be synchronized, they must know the run-time identifier of each other. Several methods are available to accomplish this, as described in the following:

- Any task can determine its own runtime identifer by using the Self Identification SVC.
- When one task places another task in execution by using the Execute Task supervisor call, the calling task is informed of the run-time ID of the called task. If the called task needs to know the run-time ID of the calling task, it must be passed from the calling task to the called task as a parameter.
- If more comprehensive communications is required among cooperating tasks, a shared procedure segment may be used. The shared procedure segment is preferable to system common memory for cooperating tasks, since the size is not limited by a system generation parameter.

The following is an example of an SVC call block that performs the Activate Suspended Task call for a task with a run-time ID of >3C:

```
SCBB    BYTE     07    ACTIVATE SUSPENDED TASK OPCODE
ERRC    BYTE     00    SET BYTE ONE TO ZERO
TID     BYTE    >3C    RUNTIME ID OF TASK TO BE ACTIVATED
```

Within the procedure portion of the calling task, the following statement is used to perform the operation:

```
SVC          @SCBB
```

If task >3C had previously been suspended by an Unconditional Wait call, it would be activated by the preceding statement. If task >3C had not been suspended, but is suspended by an Unconditional Wait statement subsequent to the preceding call, it would be immediately reactivated.

8.2.5 >09 — Do Not Suspend SVC

The Do Not Suspend SVC causes the system to override the time slice for the calling task by inhibiting the system from suspending the task. The task may suspend itself by executing an I/O SVC, or a Time Delay, Wait for I/O, or an Unconditional Wait SVC. The call block contains two bytes, and need not be aligned on a word boundary. Byte 0 contains the code, and byte 1 contains 0 or a positive number. When byte 1 contains 0, the task will not be suspended for 200 ms. When byte 1 contains a positive number, the task will not be suspended for that number of system units (50 ms).

Suspension of a task may be inhibited for a period of from 50 ms. to 12.750 seconds (1 to 255 50-ms periods).

The following example shows a coded call block for a Do Not Suspend SVC:

```
SCB     BYTE    9, 15     INHIBIT SUSPENSION OF CALLING TASK
*                         FOR 750 MS
```

The following call is an example to perform the Do Not Suspend SVC using the preceding call block:

```
SVC     @SCB
```

Use the Do Not Suspend SVC to inhibit termination of a task at the end of a current time slice instead of a LIMI instruction with an operand of zero.

When a task manipulates a data structure that is used by several tasks, the task should complete its alterations to the data structure before any of the tasks that use the data execute again. The Do Not Suspend supervisor call allows such a task to lock out other tasks while changing the data.

8.2.6 >0E — Activate Time Delay Task SVC

The Activate Time Delay Task SVC activates a specified task that is in a time delay state. A task enters the time delay state when a Time Delay SVC is issued within the task. The call block for the Activate Time Delay SVC consists of three bytes and does not require alignment on a word boundary.

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >0E.
1		Initialize to zero. System returns any error.
2		Runtime ID of task to be activated.

The error code returned is the task state. The following is an example of an SVC block coded to activate task >1B:

```
SCBH    BYTE    >0E     SUPERVISOR CALL OPCODE
ERRC    BYTE    0       SET BYTE ONE TO ZERO
TRID    BYTE    >1B     TASK IDENTIFIER (RUNTIME)
```

Code the following statement in the procedure portion of the calling task to perform the Activate Time Delay Task functions:

```
SVC          @SCBH
```

8.2.7 >11 — Change Priority SVC

The Change Priority supervisor call allows the calling task to change its own priority. This is valuable in a situation where a task that normally runs at low priority needs to perform a critical operation. Use of this call allows a task to change its priority to a higher or lower level.

The call block for the Change Priority SVC consists of two bytes and does not need to be aligned on a word boundary. Byte zero of the call block contains the SVC code >11 and byte one contains the new priority level. On return from the SVC call, the priority in effect prior to the call is returned in byte one. This priority code can be 0, 1, 2, or 3 if the task is not real-time task priority, and between 1 and 127 (inclusive) if the task is real-time priority. Real-time priority is indicated by setting the high-order bit. These priorities are explained following the example. (Priority 4, which automatically alternates between priorities 1 and 2 is not a valid priority level for this SVC.)

An example of a call block required to change the calling task's priority level to real-time priority >80 is as follows:

```
SCBG   BYTE   >11    CODE OF THE CHANGE PRIORITY CALL
PRI    BYTE   >D0    SET PRIORITY TO REAL-TIME 80
```

Within the procedure portion of the task, the following statement causes the Change Priority call to be performed:

```
SVC    @SCBG       PERFORM EXTENDED OPERATION 15 USING
                   THE CALL BLOCK AT LOCATION SCBG
```

Specification of an illegal priority will result in an error code of >80 being returned in byte 1 of the call block.

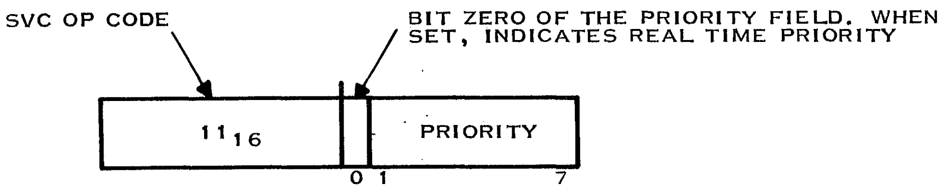
The DX10 operating system requires that each task have a defined priority level. There are 132 priorities available: Level 0 (highest), 127 real-time priorities (denoted R1-R127), level 1, 2, and 3 (lowest). Also, priority four is a dynamic priority which is level one while in the interactive mode, and otherwise is level two.

Level zero is intended for the most critical system functions and is thus reserved for DX10 internal use. The remainder of the system tasks are appropriately distributed among the lower priorities with regard to their relative importance.

Real-time priorities provide the user the capability to supersede all except the most important system tasks. For these applications which require an expeditious access to the CPU, DX10 will forgo some routine maintenance of system duties in an effort to schedule real-time tasks. The actual priority strength of the real-time priority levels 1-127 falls between priority level 0 and 1 of the nonreal-time priority levels.

Priorities one, two, three and four are designed to satisfy the requirements of most installations. Priority level one gives quick response for programs which interact with the user's terminal, while priority level two is adequate for programs which perform multiple disk accesses. Both levels one and two are used mainly by programs which require some interaction with a user. For programs which do both user interaction and multiple disk access, priority four will automatically switch between priorities one and two as the program executes. This is advantageous to the overall system in that programs that require user attention can be serviced quickly. Priority level three is for batch execution, which requires no user interaction.

Bit zero of the priority field, when set, indicates real-time priority as shown below:



8.2.8 >14 — Load an Overlay SVC

The Load Overlay SVC is used to load a specified overlay from disk into memory. Within the call block, the user specifies an overlay number, the LUNO of the overlay's program file, and the address at which the overlay is to be loaded. The call block consists of seven bytes and is aligned on a word boundary. The bytes within the call block have the following meanings:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >14.
1		Initialize to zero. System returns any error.
2-3		Contain the optional user specified load address for the overlay. This parameter is used along with information supplied when the overlay was installed to determine the load address and whether relocation is to be performed. Certain restrictions apply, as discussed in the following paragraphs.
4-5		Contain the number of the overlay to be loaded. Specify an overlay number that is less than or equal to 255 (decimal).
6		Contains the user-specified LUNO of the program file where the overlay resides. This LUNO must not be open when the call is issued. Two special-case values are as follows: If the LUNO field is zero, the overlay is loaded from the system program file. If the LUNO field is >FF, the overlay is loaded from the same program file as the calling task. In this case, do not release LUNO >FF.

If bytes two and three of the call block zero or equal to the natural load address of the overlay (the natural load address is determined when the overlay is linked), the overlay is loaded at the natural load address without relocation.

If bytes two and three of the call block are not equal to either zero or the natural load address, the overlay is loaded at the address specified in bytes two and three. Relocation is performed only if relocation was specified when the overlay was installed.

If relocation is to be performed, the user-supplied load area must provide space for the overlay and the relocation bit map, which is loaded immediately following the overlay. The size of the relocation bit map, in bytes, is calculated as follows:

$$SBM = 2 \times \left[\frac{SOV + 31}{32} \right]$$

where:

SBM is the size of the bit map and SOV is the size of the overlay, in bytes. The fraction is rounded up to the next integer.

The total size of the user load area (in bytes) when relocation is to be performed is SOV + SBM.

The following example shows the call block to load overlay number >6A from LUNO >1B into memory at the natural overlay load address.

```
        EVEN
SCBD    BYTE    >14      CALL CODE
ERRC    BYTE     0       ERROR CODE
LADD    DATA     0       NO RELOCATION
OVNM    DATA     6A      OVERLAY NUMBER 96
LNUM    BYTE     1B      LUNO OF PROGRAM FILE OVERLAY IS ON
```

Code the following statement in the procedure portion of the calling task to implement this SVC:

```
SVC             @SCBD
```

8.2.9 >16 — End of Program SVC

The End of Program SVC is identical to the End of Task SVC, except that the call code specified in the call block is >16. This call is supported only for compatibility with previous systems, and its use is not recommended.

8.2.10 >17 — Get Parameters SVC

The Get Parameters SVC obtains the task parameters passed to the calling task by the Execute Task or Scheduled Bid Task SVCs. The SVC call block must be aligned on a word boundary and contain six bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >17.
1		Initialize to zero. System returns any error.
2-5		The system places the parameters specified in the Execute Task or Scheduled Bid Task call into these four bytes.

The following is an example of the call block for the Get Parameters SVC:

```
       EVEN
SCBI   BYTE   >17    SVC OPCODE
ERRC   BYTE     0    ERROR BYTE
PAR1   DATA     0    SET TO ZERO
PAR2   DATA     0    SET TO ZERO
```

An example of the call to perform the Get Parameters function is as follows:

```
SVC    @SCBI
```

8.2.11 >1F — Scheduled Bid Task SVC

The Scheduled Bid Task SVC activates a nonreplicatable task at a specified time. The task must be installed on the system program file.

The Scheduled Bid Task SVC operates in one of two modes, depending on whether the task is scheduling itself or another task. If the task issuing the Scheduled Bid Task call is scheduling itself, then the call functions as a time delay. Control does not return to the calling task until the specified date and time. If the task being scheduled is not the calling task, control returns to the calling task after the requested task has been functionally placed active with a time delay set to start execution at the specified time and date. For reliable results, the specified time should be more than the current time by several seconds.

The call block for the Scheduled Bid Task SVC must be aligned on a word boundary, and requires 14 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >1F.
1		Initialize to zero. System returns any error.
2		Contains the user specified installed identifier.
3		The value of the year in which the bid task is to be activated. The year is specified relative to 1900 A.D. For example, the year 1978 is expressed as 78 (decimal).
4-5		Contain the Julian date on which the task is to be activated. For example, the date Feb. 10 is expressed as 41 (decimal).
6		Contains the user specified hour at which the task is to be activated. The hour is based on twenty-four hour notation. For example, 11:00 PM is expressed as 23 (decimal). The value must be less than or equal to 23 (decimal).
7		Contains the user specified minute at which the task is to be activated. The minute must be less than or equal to 59 (decimal).
8		Contains the user specified second at which the task is to be activated. The second must be less than or equal to 59 (decimal).
9		Reserved. Initialize to zero.
10-13		These bytes are optional, and if specified by the user, contain parameters that are to be passed to the bid task.

The following example shows the call block for a Scheduled Bid Task operation to activate task >3D at 11:59:59 on February 10, 1982. In the example, the parameters ABCD are passed to the bid task.

```
        EVEN              ALIGN ON WORD BOUNDARY
SCBC    BYTE    >1F       OPCODE FOR SCHEDULED BID TASK
ERRC    BYTE     00       SET BYTE ONE TO ZERO
TID     BYTE    >3D       TASK ID
YEAR    BYTE     82       YEAR THE TASK IS TO BE ACTIVATED
DAY     DATA     41       JULIAN DATE OF THE ACTIVATION DATE
HOUR    BYTE     23       HOUR THE TASK IS TO BE ACTIVATED
MIN     BYTE     59       MINUTE AT WHICH TASK IS TO BE ACTIVATED
SEC     BYTE     59       SECOND AT WHICH THE TASK IS TO BE ACTIVATED
        BYTE     00       SET BYTE NINE TO ZERO
MSG     TEXT    'ABCD'    PASS 'ABCD' TO TASK
```

Code the following statement into the procedure portion of the calling task to perform the Scheduled Bid Task SVC using the information at the memory location specified by SCBC:

```
SVC     @SCBC
```

8.2.12 >2B — Execute Task SVC

The Execute Task SVC is used to initiate the execution of a task that has been installed on any program file. If the task specified in the call is already active and was defined as being replicatable (a task is defined to be replicatable when it is installed), another copy of the task is placed in execution. The replicated task can share procedures with previous activations of the task. If the call is issued for a task that is active but is not replicatable, the system returns an error to the calling task.

The call block for the Execute Task SVC is twelve bytes in length and must be aligned on a word boundary. The entries in the supervisor call block are defined as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >2B.
1		This byte is used by the system to return an error code, if necessary.
2		Contains the ID of the task to be executed. The user specifies the installed ID of the task and the system returns the run-time ID in this byte. The system returns the run-time ID of the currently executing task if an attempt is made to replicate a non-replicatable task; an error code of >FA is also returned in byte 1.
3		Contains user specified flags with the following meanings:
	0–3	Set to zero.
	4	When set to one, this bit causes the calling task to be terminated immediately after issuing the Execute Task SVC; all other user flags are ignored. When zero, the calling task is not terminated. This is the chaining flag.
	5	When set to one, this bit indicates that the task specified in byte 2 is a controlled task. This flag is used in conjunction with the system debug utility, which is described in Section 11.
	6	When set to one, this bit forces the initial state of the executed task to be unconditionally suspended. This flag is also used in conjunction with the system debug utility.
	7	Setting this bit to one causes DX10 to suspend the calling task until the called task (specified in byte 2) has terminated.
4–7		These bytes are used to pass user specified parameters to the called task. The called task must issue a Get Parameters SVC (>17) to obtain the parameters.

Byte	Bit	Meaning
8		Contains the user specified ID of the station (terminal) with which the called task is to be associated. If the user specifies 0 (zero), the called task will be associated with the same station as the calling task. If the user specifies >FF, the called task will not be associated with a station.
9		Contains the user specified logical unit number (LUNO) assigned to the program file on which the called task is installed. If the user specifies zero, the system assumes that the single system program file is to be used. If the user specifies >FF, the program file of the calling task applies.
10		Set to zero.
11		Set to zero.

The chaining flag (bit 4 of byte 3) works as follows:

Suppose task A bids task B using the Execute Task SVC, with bit 7 of byte 3 set in the SVC call block. Task A is suspended in state >17 until task B terminates. However, if task B then bids task C with the chaining flag set in the Execute Task SVC call block, task A is reactivated only when task C terminates. Similarly, if task C then bids task D with the chaining flag set in the Execute Task SVC call block, task A is reactivated only when task D terminates. In other words, task A is reactivated only when the last task in the "task chain" has terminated. In most applications, task A will be SCI.

The following is an example of an Execute Task SVC.

```
        EVEN             FORCE ALIGNMENT ON A WORD BOUNDARY
SCBA    BYTE    >2B      OPCODE FOR EXECUTE TASK
ERRC    BYTE    >00      SET BYTE ONE TO ZERO
T1D     BYTE    >3B      INSTALLED ID OF TASK TO BE EXECUTED
FLG1    BYTE    >01      CALLING TASK SUSPENDED UNTIL CALLED TASK
                           TERMINATES
PAR     TEXT    'HELP'   PASS 'HELP' TO CALLED TASK
FLG2    BYTE    >FF      TASK ASSOCIATED WITH NO STATION
LUN     BYTE    >1A      LUNO OF THE PROGRAM FILE
        BYTE    00,00    SET BYTES 10 AND 11 TO ZERO
```

Within the procedure portion of the calling task, the following statement is used to initiate the Execute Task SVC:

```
SVC     @SCBA
```

Issuing the preceding call causes task >3B to be loaded from the program file and executed. If the task issues a Get Parameter call, it is passed the ASCII characters HELP.

8.2.13 >2E — Self-Identification SVC

The Self-Identification supervisor call allows the calling task to obtain its run-time ID, its installed ID, and the ID of the terminal station with which it is associated. If it is not associated with any station the system returns a station ID of >FF.

The call block must be aligned on a word boundary, and requires six bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >2E.
1		Receives the runtime ID.
2		Receives the installed ID.
3		Receives the task's associated station ID.
4-5		Reserved. Set to zero.

The following example shows how to code the call block for this SVC:

```
        EVEN
SCBJ    BYTE    >2E    SVC OPCODE
RTID    BYTE     0     SYSTEM RETURNS RUNTIME ID HERE
INID    BYTE     0     SYSTEM RETURNS INSTALLED ID HERE
STID    BYTE     0     SYSTEM RETURNS STATION ID HERE
        DATA     0     SET TO ZERO
```

The call to request the operation defined by the preceding block is written as follows:

```
SVC     @SCBJ
```

8.2.14 >2F — End Action Status SVC

The End Action Status supervisor call is used to determine the Workspace Pointer, Status, and Program Counter register contents when an error causes abnormal terminating of a task. In addition, the system returns the code of the error that caused termination. This call is issued during end action execution. End action execution occurs when an error aborts the task, causing the system to transfer control to the end action routine specified by the user in the third word of the task.

The SVC call block for the End Action Status call must be aligned on a word boundary, and consists of 10 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >2F.
1		Initialize to zero. System returns any error.
2–3		Contains the Workspace Pointer register contents.
4–5		Contains the Program Counter register contents.
6–7		Contains the Status register contents.
8–9		Reserved. Set to zero.

The following example shows a coded call block for this SVC.

```
        EVEN
SCBK    BYTE    >2F     SVC OPCODE
ERRC    BYTE    0       ERROR CODE RETURNED HERE
WPREG   DATA    0       WORKSPACE POINTER REGISTER CONTENTS
PCREG   DATA    0       PROGRAM COUNTER REGISTER CONTENTS
STREG   DATA    0       STATUS REGISTER CONTENTS
        DATA    0       SET TO ZERO
```

The following is an example of the code to initiate this SVC, using the preceding block:

```
SVC     @SCBK
```

8.2.15 >31 — Map Program Name to ID SVC

Use the Map Program Name to ID SVC to determine the installed ID of any procedure, task, or overlay, or to determine the module name. If you code the call block to return the installed ID, you must provide the module name. If you code the call block to return the module name, you must supply the installed ID. The call block must be aligned on a word boundary, and requires 16 bytes, as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >31.
1		Set to 0 (zero) by the user. If an error occurs, the system returns an error code in this byte.

Byte	Bit	Meaning
2		User-specified flags with the following definitions:
	0–1	Indicates whether the data is requested for a task, a procedure or an overlay: 00 a task's ID or name is requested. 01 a procedure's ID or name is requested. 10 an overlay's ID or name is requested.
	2	Indicates what type of data is requested. 0 indicates the ID is requested. 1 indicates the program name is requested.
	3-6	Not used. Set to zero.
	7	Specifies whether the LUNO assigned to the program file is open or not open. 1 indicates the LUNO is open. 0 indicates the LUNO is not open.
3		Reserved. Set to zero.
4–11		These bytes contain the name of the task, procedure or overlay. If you are requesting the ID, you supply the name in these bytes. If you are requesting the name, the system returns it in these bytes. Assume trailing blanks.
12		Specify the LUNO of the program file where the task, procedure, or overlay resides. Zero indicates the single system program file is used.
13		These bytes contain the installed ID of the task, procedure or overlay. If you request the name, supply the ID in these bytes. If you request the installed ID, the system returns it in these bytes.
14–15		Reserved. Set to zero.

The following example illustrates a call block to determine the program name of a task (>1B) installed on a program file with the LUNO assignment >2C.

```
        EVEN
SCBL    BYTE    >31     SVC OPCODE
ERRC    BYTE     0      ERROR BYTE, SET TO ZERO
FLGA    BYTE    >20     SPECIFY REQUEST FOR TASK NAME
        BYTE     0      INITIALIZE TO ZERO
NAME    BSS      8      BLOCK OF EIGHT BYTES FOR NAME
LNUM    BYTE    >2C     LUNO
TID     BYTE    >1B     TASK ID
        DATA     0
```

An example of code required to determine the installed ID of a procedure installed on the program file assigned to LUNO >23 and with a program name of CALCFOUR, is as follows:

```
        EVEN
SCBM    BYTE    >31     SVC OPCODE
ERRC    BYTE     0      ERROR BYTE, SET TO ZERO
FLGA    BYTE    >40     PROCEDURE ID REQUESTED
        BYTE     0      RESERVED, SET TO ZERO
NAME    TEXT    'CALCFOUR'  PROCEDURE NAME
LNUM    BYTE    >23     LUNO NUMBER
TID     BYTE     0      ID RETURNED HERE
        DATA     0
```

The following calls are examples of those written to perform the Map Program Name to ID supervisor call using the preceding blocks:

```
SVC     @SCBL

SVC     @SCBM
```

8.2.16 >35 — Poll Status of Task SVC

The Poll Status of Task SVC determines the status of a special task within the set of tasks assigned to a specified terminal. Tasks are assigned to a specific terminal when they are placed in execution by the Execute Task supervisor call.

NOTE

If the task was terminated using a Kill Task SVC (code >33), it is possible for the Poll Status call to return a code of >FF (task is not in the system) while the task's LUNOs are still assigned. This is due to a time interval between the time the task is removed from the active task queue and when its LUNOs are closed and released.

The call block for the Poll Status of Task must be aligned on a word boundary, and requires 8 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC code. Must be >35.
1		Initialize to zero. System returns any error.
2		Contains user-specified flags as follows:
	0	Set to 1 to indicate that a station number is specified in byte 4. Set to zero if you want to poll only tasks executed from the same station as this SVC.
	1–7	Reserved. Set to zero.
3		The system returns the task state code for the specified task.
4		If byte 2, bit 0 is set to 0, set this byte to zero, since no station number is needed. If byte 2, bit 0 is set to 1, then specify the station number associated with the set of tasks to be polled. If you want the status of a task no matter what its station, place >FF in this byte.
5		Specify the runtime ID of the task whose status you want reported.
6–7		Reserved. Set to zero.

The following is an example of a Poll Status of Task call block to determine the status of a task with a run-time ID of >3F and that is assigned to terminal eight.

```
        EVEN
SCBG    BYTE    >35     SVC OPCODE
ERRC    BYTE     0      SET ERROR BYTE TO ZERO
FLAG    BYTE    >80     STATION NUMBER SPECIFIED
TSKC    BYTE     0      TASK STATE CODE RETURNED HERE
STAN    BYTE     8      STATION NUMBER
TID     BYTE    >3F     TASK RUNTIME ID
        DATA     0      SET BYTES 6 AND 7 TO ZERO
```

The Poll Status function using the above call block is called by using the following statement within the procedure portion of the calling task:

```
SVC     @SCBG
```

Note that the calling task has to obtain the task state from byte 3 of the call block.

8.2.17 >3E — Reset End Action SVC

The Reset End Action SVC resets a calling task's end action capability if the run ID specified in the call block is zero; otherwise, the task ID specified in the call block will have end action reset. If the task specified already has end action set, no error code is returned.

DX10 allows a task to take end action only one time. The Reset End Action SVC allows a task to take end action more than one time. The task that requires more than one attempt at end action may issue this SVC, or a controlling task may reset end action for another task. The Reset End Action SVC must be executed each time end action is taken and will not reset end action if the task has been terminated via a Kill Task request.

The supervisor call block for the Reset End Action must be aligned on a word boundary, and requires 4 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >3E.
1		Initialize to zero. System returns any error.
2		System returns the task's runtime ID.
3		Reserved. Set to zero.

The following example shows a coded call block for this SVC.

```
        EVEN
SCBK    BYTE    >3E     SVC OPCODE
ERRC    BYTE     0      ERROR CODE RETURNED BY SYSTEM
RTID    BYTE     0      SYSTEM TASK RUN ID, OR ZERO
        BYTE     0      SET TO ZERO
```

The following is an example of the code to initiate the Reset End Action call using the preceding block:

```
SVC     @SCBK
```

8.3 MEMORY CONTROL

Memory Control SVCs allow the calling task to expand its memory space, to access system common memory, to contract its memory space, or to release system common memory. When using any of these SVCs, you should take certain considerations into account.

A task installed in DX10 can access three distinct portions of memory by using the three pairs of memory mapping address registers that are assigned to each task. The task itself can use one, two, or all three of these registers. A program installed as one object module, consisting of a task division and a procedure division, uses one map register; a program installed as a task segment with a separate procedure segment uses two of the map registers; and a program installed as a task segment with two separate procedure segments uses all three of the map registers. The use of the registers and the distinction between separate tasks and procedures is transparent to the user since the link editor provides all the links between portions of a task. However, use of Memory SVCs affects the contents of these registers, and so have certain restrictions.

Calls to expand or contract the actual task memory do not require an additional map register; they simply modify the register that defines the limits of the task. Note, however, that the call to access system common memory uses one of the registers. The call to return the system common memory releases the map register used. If the task itself uses all three registers, no calls may be made to access system common memory. The user should also note that while system common memory is mapped into the task's address area, no calls can be issued to expand or contract the task's actual memory size.

NOTE

DX10 does not keep the history of Get and Release memory calls. For this reason, it is not possible to release a specific block of memory, except at the high end of memory when the task itself remembers the specific block acquired.

8.3.1 >10 — Get Common Data Address SVC

This SVC is included for compatibility with previous systems and is not recommended for new designs. The preferred technique is to use a common procedure element, since procedures may be linked with the task to resolve references.

The Get Common Data Address SVC obtains the address of the beginning of the intertask common memory area, and to return that address in task workspace register 9. The size of the intertask common memory area is returned to the calling task in its workspace register 8. The size of the intertask common memory area is a system parameter specified during system generation.

To properly use the Get Common Data Address SVC, you must understand that each task is allowed to use only three segments of memory at one time. This is the basis of task segmentation, as described in Section 2.

The SVC call block for the Get Common Data Address SVC does not need to be aligned on a word boundary, and requires only two bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >10.
1		Initialize to zero. The system returns any error. The possible codes returned are the standard task error codes, plus >FF. The >FF code indicates that the three allowable segments of memory were already in use when this call was issued, or that a common area was not included during system generation.

The following is an example of a Get Common Data Address call block:

```
SCBD    BYTE    >10     SVC OPCODE
ERRC    BYTE     0      ERROR BYTE SET TO ZERO
```

The following code initiates the Get Common Data Address SVC:

```
SVC     @SCBD
```

8.3.2 >12 — Get Memory SVC

The Get Memory SVC increases the amount of memory allocated to the calling task. You must specify the number of contiguous 32-byte blocks of memory that you wish to obtain for the task. The address of the first 32-byte block obtained is returned to the calling task in Workspace Register 9.

As the SVC executes, the task is rolled out and then rolled back into memory. The new memory area equals the original size of the task plus the additional memory requested by the Get Memory call.

You cannot issue the Get Memory SVC when the task is memory-resident, or when system common memory is mapped into the task's address space.

You can issue any number of Get Memory SVCs, as long as the maximum amount of memory available to an individual task is not exceeded. This maximum is 64K bytes. You can issue any number of Get Memory SVCs within a task as long as the preceding requirements are met.

The call block for the Get Memory SVC must be aligned on a word boundary, and requires four bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >12.
1		Initialize to zero. System returns any error.
2		Specify the number of 32-byte blocks of memory that you want allocated to the calling task.

The following example shows a call block for a Get Memory SVC that requests 16 contiguous 32-byte blocks of memory to the calling task:

```
        EVEN
SCBB    BYTE    >12     SVC OPCODE
ERRC    BYTE     0      INITIALIZE ERROR CODE BYTE TO ZERO
BLKC    DATA     16     NUMBER OF MEMORY BLOCKS REQUESTED
```

The following example shows the code required to initiate the Get Memory SVC using the preceding example block.

```
SVC     @SCBB
```

8.3.3 >13 — Release Memory SVC

The Release Memory SVC is used to return to the system all task memory above the address specified in workspace register 9. The Release Memory call cannot be issued if the task is memory-resident nor can it be issued if the task currently has system common memory mapped into its address space. The Release Memory call modifies the task's mapping register to address only the remaining task memory area.

The call block for the Release Memory SVC must be aligned on a word boundary, and requires 4 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >13.
1		Initialize to zero. System returns any error.
2–3		Not used. Set to zero.

The following example shows a Release Memory call block that releases memory beyond the address >8400 in the task area.

```
        EVEN
SCBC    BYTE    >13     SVC OPCODE
ERRC    BYTE     0      INITIALIZE BYTE ONE TO ZERO
BLKC    DATA     0
```

The following code initiates the Release Memory SVC, using the call block in the preceding example.

```
LI      R9,>8400
SVC     @SCBC   RELEASE MEMORY ABOVE ADDRESS >8400
```

NOTE

Once the call in this example is executed, an attempt to release memory higher than >8400 would result in a fatal task error.

8.3.4 >1B — Return Common Data Address SVC

The Return Common Data address SVC releases the system common memory obtained by the Get Common Data Address SVC, and allows the task to request another segment of memory. Additional segment requests are limited by the memory segments constraints discussed earlier in this section.

The call block for the Return Common Data Address SVC requires one byte, containing the SVC code (>1B), and does not need to be aligned on a word boundary. An example of the call block is as follows:

```
SCBE    BYTE    >1B     SVC OPCODE
```

The code required to implement this call block is as follows:

```
SVC     @SCBE
```

8.4 INTERTASK COMMUNICATIONS

The Intertask Communications SVCs allow you to transfer data between tasks using a dynamic memory buffer. The data transfer usually occurs by one task placing data in the DX10 dynamic memory area to later be retrieved by another task.

To avoid duplication of message subqueue IDs in byte three of the SVC call block, use the run-time task ID of one of the tasks involved in the data transfer. A combination of the self-identification (>2E), Execute Task (>2B), and Get Parameters (>17) SVCs can be used to accomplish identification of the subqueues with one of the task's run-time IDs. This is the mode of identification used by the TI Sort/Merge and the TI COBOL Sort function.

The two available intertask communications SVCs are Putdata and Getdata.

8.4.1 >1C — Putdata SVC

The Putdata SVC transfers messages from a user buffer to channel or "subqueue" in the DX10 dynamic memory area for subsequent retrieval by a Getdata SVC. All messages with the same ID are placed on one subqueue. That subqueue is then identified by the ID of the messages it contains. The messages are placed in the subqueues in a first-in, first-out manner, so that the oldest message in the subqueue is obtained first when a Getdata call is issued with that identifier. The Putdata operation is unsuccessful when a message buffer cannot be allocated.

The call block for the Putdata SVC must be aligned on a word boundary, and requires 12 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >1C.
1		Initialize to zero. System returns any error. If the system cannot allocate a buffer for the message, the system returns >FF.
2		Reserved. Initialize to zero.
3		Specify the message identifier in this byte.
4-5		Specify the address of the task buffer containing the message. The address must begin on a word boundary.
6-7		Not used. Set to zero.
8-9		Specify the length of the buffer that contains the message. The length must be an even number of bytes.
10-11		Reserved. Initialize to zero.

The following example of a Putdata call block requests that a message be written from the task buffer at location BUFF2, which has a maximum length of 80-characters. The message is given an identifier of >1F.

```
        EVEN
SCBG    BYTE    >1C     SVC OPCODE
ERRC    BYTE     0      ERROR BYTE. INITIALIZE TO ZERO
RESV1   BYTE     0      INITIALIZE BYTE TWO TO ZERO
MID     BYTE    >1F     IDENTIFIER FOR THE MESSAGE
BADR    DATA     BUFF2  BUFFER ADDRESS
RESV2   DATA    0
BLGT    DATA    80      LENGTH OF THE BUFFER
RESV3   DATA    0       INITIALIZE BYTES TEN AND ELEVEN TO ZERO
```

The following code initiates the Putdata function using the preceding example call block:

```
SVC     @SCBG
```

8.4.2 >1D — Getdata SVC

The Getdata SVC retrieves a message from the DX10 dynamic memory area and moves the message to the specified user buffer. You must supply an identifier (ID) in the call block to specify which channel or subqueue contains the message. Note that messages are queued in a first-in, first-out order, so that the SVC always returns the oldest message on the specified queue. The returned message is deleted from the subqueue. If no message with the specified ID is found, the system returns an error.

You can optionally delete all messages on a specified subqueue using this SVC. This option does not return any messages.

The call block for the Getdata SVC must be aligned on a word boundary, and requires 12 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >1D.
1		Initialize to zero. System returns any errors. If no message with the specified ID is found, the system returns >FF.
2		User-specified flags with the following meanings:
	0	If set to one, every message in the subqueue indicated by the ID is deleted and no message is returned. If set to zero, a message is deleted from the subqueue only when it is returned.
	1–7	Reserved. Initialize to zero.
3		The user specified ID of the subqueue containing the message.
4–5		Specify the address of the buffer where the message is to be placed. Must be on a word boundary.
6–7		Specify the maximum length of the buffer where the message is to be placed. Must be an even number of bytes.
8–9		Contains the actual length of the message. The system returns this value. The length is always less than or equal to the buffer length. Messages longer than the buffer are truncated and the truncated part is lost.
10–11		Reserved. Initialize to zero.

The following example of a Getdata call block requests that a message with a subqueue ID of >7A be returned to the task in the buffer location BUFF1, with a maximum buffer length of 80-characters.

```
        EVEN
SCBF    BYTE   >1D     SVC OPCODE
ERRC    BYTE    0      ERROR BYTE. INITIALIZE TO ZERO
FLG1    BYTE    0      DO NOT PURGE SUBQUEUE
MID     BYTE   >7A     MESSAGE SUBQUEUE ID
BADR    DATA    BUFF1  ADDRESS OF BUFFER THAT IS TO RECEIVE MESSAGE
MLGT    DATA   80      MAXIMUM LENGTH IS 80-CHARACTERS
ACTL    DATA    0      INITIALIZE BYTES EIGHT THROUGH ELEVEN TO ZERO
RESV    DATA    0
```

The following code initiates the Getdata operation using the preceding example call block:

```
SVC     @SCBF
```

8.5 DATA CONVERSION SERVICES

DX10 supports SVCs that provide data conversion services such as binary-to-decimal, decimal-to-binary, binary-to-hexadecimal, and hexadecimal-to-binary conversion. The following paragraphs describe each of these SVCs and provide examples of the associated call blocks.

8.5.1 >0A — Convert Binary-to-Decimal SVC

The Convert Binary-to-Decimal SVC converts a binary value located in workspace register 0, to its equivalent decimal value, represented in ASCII. The SVC call block for the Convert Binary-to-Decimal SVC requires 8 bytes, and does not have to be aligned on a word boundary. The call block is as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >0A.
1		Reserved. Initialize to zero.
2		Contains the ASCII sign. If the converted value is negative, the system returns an ASCII minus sign (>2D) in this byte. If the converted value is zero or positive, an ASCII blank (>20) is returned in this byte.
3–7		Contains the ASCII decimal value. The system places the converted value into these bytes, right justified with leading blanks.

The following example shows a call block coded to perform a Convert Binary-to-Decimal operation:

```
SCBH    BYTE    >A     SVC OPCODE
ERRC    BYTE     0     SET ERROR BYTE TO ZERO
WRD1    BYTE     0,0   FIRST TWO BYTES FOR DECIMAL VALUE
WRD2    BYTE     0,0   NEXT TWO BYTES FOR DECIMAL VALUE
WRD3    BYTE     0,0   LAST TWO BYTES FOR DECIMAL VALUE
```

The following code initiates the Convert Binary-to-Decimal SVC using the preceding example call block:

```
SVC     @SCBH
```

The following examples show several binary values in register 0 (noted in hexadecimal), along with the results returned by the Convert Binary-to-Decimal SVC. The values are returned in bytes 2–7 of the call block:

Register 0	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
>0001	>20	>20	>20	>20	>20	>31
>7FFF	>20	>33	>32	>37	>36	>37
>FFFF	>2D	>20	>20	>20	>20	>31

8.5.2 >0B — Convert Decimal-to-Binary SVC

The Convert Decimal-to-Binary SVC converts a decimal value, represented in ASCII in bytes 2-7 of the SVC call block, to a binary value, and places that binary value in the task workspace register 0. The SVC call block requires 8 bytes and need not be aligned on a word boundary. The bytes in the call block have the following meanings:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >0B.
1		Initialize to zero. System returns any error. If one of the characters specified in bytes 2-7 is not valid, the system returns >FF.
2		Specify the sign of the value to be converted. Allowable entries are: >2B — The ASCII representation of a plus sign (+). >2D — The ASCII representation of a minus sign (−). >20 — The ASCII representation of a blank ( ), which indicates the value is positive or zero. >30 — The ASCII representation of a zero (0), which indicates that the value is positive or zero.
3-7		Specify the ASCII representation of the value to be converted. Right-justify the entry, using leading blanks.

The following example shows an SVC call block for converting the decimal value 378 to binary, and placing the value in workspace register 0.

```
SCBI    BYTE   >0B      SVC OPCODE
ERRC    BYTE    0       ERROR CODE. INITIALIZE TO ZERO
SIGN    BYTE    '+'     PLUS SIGN TO INDICATE POSITIVE OR ZERO
                          VALUE
CDAT    TEXT   '00378'  CONVERT 378 DECIMAL TO BINARY
```

The following example initiates this SVC using the preceding call block.

```
SVC     @SCBI
```

The following example shows decimal values converted to binary. The decimal values are placed in bytes 2-7 using the hexadecimal ASCII representation for each digit. The binary value returned is shown in register 0 in hexadecimal representation. Byte 1 receives any error generated.

Value	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	R0	Byte 1
+00001	>2B	>20	>20	>20	>20	>31	>0001	00
1A	>20	>20	>20	>20	>31	>41	Undefined	>FF
032767	>30	>33	>32	>37	>36	>37	>7FFF	00
−00001	>2D	>30	>30	>30	>30	>31	>FFFF	00

8.5.3 >0C — Convert Binary-to-Hexadecimal SVC

The Convert Binary-to-Hexadecimal SVC converts a binary value in the calling task's workspace register 0, to the ASCII representation of the equivalent hexadecimal value. The SVC call block requires 6 bytes, and does not need to be aligned on a word boundary.

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >0C.
1		Initialize to zero. System returns any error.
2-5		Contains the hexadecimal value returned by the system.

The following is an example of a Convert Binary-to-Hexadecimal call block. The binary value to be converted is in workspace register 0.

```
SCBJ    BYTE    >0C     SVC OPCODE
ERRC    BYTE     0      ERROR CODE
CNU1    BYTE     0, 0   FIRST TWO BYTES OF CONVERTED VALUE
CNU2    BYTE     0, 0   NEXT TWO BYTES OF CONVERTED VALUE
```

The following code initiates the Convert Binary-to-Hexadecimal operation using the preceding example call block.

```
SVC     @SCBJ
```

The following example shows typical binary values (in hexadecimal notation), and the conversion results.

Register 0	Byte 2	Byte 3	Byte 4	Byte 5
>0001	>30	>30	>30	>31
>7FFF	>37	>46	>46	>46
>FFFF	>46	>46	>46	>46

8.5.4 >0D — Convert Hexadecimal-to-Binary SVC

The Convert Hexadecimal-to-Binary SVC converts hexadecimal values, represented in ASCII in bytes two through five of the call block, to a binary value and stores it in the calling task's workspace register 0. The SVC call block requires 6 bytes, which do not need to be aligned on a word boundary.

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >0D.
1		Initialize to zero. System returns any error.
2-5		Specify the number to be converted, in ASCII.

The following example shows the call block to convert the value >03F4 to a binary value:

```
SCBH    BYTE    >0D      SVC OPCODE
ERRCD   BYTE    >0       ERROR CODE
CNV     TEXT    '03F4'   CONVERT >03F4 TO BINARY
```

The following code initiates the Convert Hexadecimal-to-Binary SVC using the preceding example call block:

```
SVC     @SCBH
```

The following example shows typical ASCII representations of hexadecimal digits converted to binary and placed in workspace register 0. Byte 1 contains any error code returned by the system.

Byte 2	Byte 3	Byte 4	Byte 5	Register 0	Byte 1
>30	>30	>30	>31	>0001	>00
>30	>30	>33	>4B	Undefined	>FF
>37	>46	>46	>46	>7FFF	>00
>46	>46	>46	>46	>FFFF	>00

8.6 SYSTEM INFORMATION

There are three SVCs that support retrieving information from the system, and sending messages to the system log. The following paragraphs define these SVCs and show examples of the associated call blocks.

8.6.1 >03 — Date and Time SVC

The Date and Time SVC obtains the time and Julian date from the system in binary form. The call block for the Date and Time SVC must be aligned on a word boundary, and requires 4 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >03.
1		Initialize to zero. System returns any error.
2–3		Address of a five-word data buffer where the system returns the time and date information.

The following is an example of a SVC call block for the Date and Time function:

```
        EVEN
SCBJ    BYTE    3          SVC OPCODE
        BYTE    0          INITIALIZES BYTE ONE TO ZERO
BUFF    DATA    DATABUF    ADDRESS OF THE FIVE-WORD AREA WHERE
*                          DATE AND TIME ARE TO BE WRITTEN.
```

The following code initiates the Date and Time function using the preceding example call block:

```
SVC     @SCBJ
```

If the date returned were July 10, 1977, and the time were 11:59:59 A.M., the five words at location DATBUF would contain the following:

YEAR	DAY	HOUR	MINUTES	SECONDS
Word 1	Word 2	Word 3	Word 4	Word 5
07B9	00BF	000B	003B	003B

The contents of DATBUF are:

Word 1 = 1977 in binary, the year
Word 2 = 191 in binary, the Julian day for July 10
Word 3 = 11 in binary, the hour from 00 to 23
Word 4 = 59 in binary, the minute
Word 5 = 59 in binary, the second

The following calling sequence converts the binary year to ASCII:

```
SCBH    BYTE    >A              SVC OPCODE
ERRC    BYTE     0              BINARY TO DECIMAL
WRD1    BYTE     0,0,0,0,0,0    CONVERTED DATA
        L1      R0,@YEAR        LOAD R0 WITH BINARY YEAR
        SVC     @SCBH           CONVERT TO DECIMAL
```

At the conclusion of the SVC, WRD1 contains 001977.

NOTE

The Date and Time SVC does not nullify the effect of the Do Not Suspend SVC (>09).

8.6.2 >21 — System Log SVC

The System Log SVC allows you to send messages to the system log. The SVC call block for the System Log SVC must be aligned on a word boundary and requires 8 bytes as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >21.
1		Initialize to zero. System returns any error.
2-3		Reserved. Initialize to zero.
4-5		Specify the address of a data buffer containing the length of the message and the message text. (The length of the message must be specified in the first byte.)
6-7		Reserved. Initialize to zero.

The following is an example of a call block to write a message at location MESS11 to the system log file:

```
        EVEN
SCBK    BYTE    >21             SVC OPCODE
ERRC    BYTE     0              INITIALIZE BYTE ONE TO ZERO
                                (ERROR CODE BYTE)
        DATA     0              INITIALIZE BYTES TWO AND THREE TO ZERO
MADR    DATA     MESS11         MESSAGE ADDRESS
        DATA     0
MESS11  BYTE     MSGEND-$-1     MESSAGE LENGTH
        TEXT     'THIS IS A MESSAGE'
MSGEND  EQU      $
```

The following code initiates the System Log SVC, using the preceding example call block:

```
SVC        @SCBK
```

The message must be preceded by a one-byte length indicator.

8.6.3 >3F — Retrieve System Information SVC

This SVC is primarily useful only to tasks written in low-level languages. The SVC obtains various types of system data but cannot modify the data. The usual call block requires 16 bytes. A short format is available for retrieving only one word. The short format is 16 bytes long, but you only need to code part of the call block. To use the short call block, set bit 0 in byte 3 to 1; zero indicates usual call block. Both formats must begin on a word boundary. A task that uses this SVC may be incompatible with future versions of DX10.

The usual format is as follows:

BYTE		
0	SVC OPCODE	RETURNED ERROR CODE
2	DATA TYPE	FLAGS
4	INDEX	
6	OFFSET	
8	LENGTH OF BUFFER	
10	LENGTH RETURNED	
12	BUFFER ADDRESS	
14	RESERVED (=0)	

2282050

The short format block requires less information than the long format. Initialize the unused bytes to zero. The short format is as follows:

BYTE		
0	SVC OPCODE	RETURNED ERROR CODE
2	DATA TYPE	FLAGS
4	INDEX	
6	OFFSET	
8	NOT USED	
10	WORD RETURNED	
12	NOT USED	
14	NOT USED	

2282051

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >3F.
1		Initialize to zero. System returns any error as follows: 0 = No error 1 = Invalid data type 2 = Invalid index, or index out of range 3 = Offset out of range
2		Data type requested: 1 = Physical Device Table (PDT) 2 = Overlay table 3 = Addressable memory (system memory)
3		User-specified flags as follows:
	0	When set to 1, indicates using short form of call block. Zero indicates normal form.
	1–7	Not used. Set to zero.
4–5		Index to desired structure. Can be from 1 to n, where n is the number of structures.

Byte	Bit	Meaning
6–7		Offset into data structure (in even number of bytes). Must be from 0 to L–1, where L is the length of the structure.
8–9		Normal call block: Length of return buffer (in bytes). Short call block: Not used.
10–11		Normal call block: Actual length returned. Short call block: Word returned.
12–13		Normal call block: Buffer address (on word boundary). Short call block: Not used.
14–15		Normal call block: Reserved. Initialize to zero. Short call block: Not used.

Use the index, bytes 4-5, to access linked data structures such as PDTs. The offset, bytes 6-7, determines the first word of the structure to retrieve, and the buffer length, bytes 10–11, determines how many words to retrieve. The short format only retrieves one word, so no buffer is needed. The word is returned in bytes 10–11 of the short format call block.

The following example shows the usual call block of the Retrieve System Information SVC:

```
        EVEN
RSISCB  BYTE   >3F      SVC OPCODE.
        BYTE   >00      ERROR CODE.
        BYTE    1       RETRIEVE A PDT.
        BYTE   >00      FLAGS (USUAL CALL BLOCK).
        DATA    5       RETRIEVE THE 5TH PDT IN THE
        DATA    0         CHAIN STARTING WITH THE FIRST
        DATA    BUFLEN    WORD FOR A LENGTH OF BUFLEN.
        DATA    0       ACTUAL LENGTH RETURNED.
        DATA    BUFADR  BUFFER ADDRESS.
        DATA    0       RESERVED.

BUFADR  BSS     72      BUFFER FOR RETURNED PDT
BUFLEN  EQU     $-BUFADR LENGTH OF BUFFER
        .
        .
        .
        SVC     @RSISCB RETRIEVE SYSTEM INFORMATION
```

The following example shows the short format of the call block:

```
        EVEN
SHORT   BYTE    >3F      SVC OPCODE.
        BYTE    >00      ERROR CODE.
        BYTE     2       GET DATA FROM OVERLAY TABLE.
        BYTE    >80      FLAGS (SHORT CALL BLOCK).
        DATA     1       INDEX (ONLY 1 OVERLAY TABLE).
        DATA    >1C      OFFSET INTO OVERLAY TABLE.
        DATA     0       IGNORED FOR SHORT CALL BLOCK.
LDTADR  DATA     0       WORD RETURNED HERE.
        DATA     0       IGNORED FOR SHORT CALL BLOCK.
        DATA     0       IGNORED FOR SHORT CALL BLOCK.
        .
        .
        .
        SVC     @SHORT   RETRIEVE WORD FROM OVERLAY
*                        TABLE.
```

9

Device I/O Supervisor Calls

9.1 DEVICE INDEPENDENT I/O

All I/O in DX10 is done to logical units rather than to devices. This allows you to treat devices the same within a given program, even though the required device may change. That is, you may wish your program to accept input from a card reader at certain times, and from a remote terminal at other times. The logical unit number (LUNO) assignment facilitates this. This is called device independent I/O, because the various I/O devices look the same to application programs. There are several SVC calls that facilitate device independent I/O, that is, they perform generically equivalent functions without any special parameters for individual devices.

The following devices can be considered equivalent, and can be accessed by a program using device-independent I/O:

911, 913, and 931 video display terminals
733 ASR/KSR data terminals
703/707 and 743 KSR data terminals
820 RO and 781 RO data terminals
840 RO data terminal
915 remote video terminal
783/785/787 data terminal
763/765 bubble memory terminals
804 card readers
588, 810, 850, 855, 2230, 2260, and 306 line printers
940 electronic video terminal/Business System terminal
LQ45 line printer

This list can be expanded to include any other device that has a device service routine (DSR) implemented as described in the *DX10 Operating System Systems Programming Guide*. During system generation, you must define all devices to the system. The preceding list describes those devices for which DSRs already exist in the operating system. For these devices, you only need to name them during system generation. For devices not on the list, you must write your own DSR, and then regenerate the system to include the device.

Even though device independent I/O operations look the same to application programs, the operations themselves do specific things to the device involved. For example, an Open Rewind SVC operation on a magnetic tape rewinds the tape to the beginning; on a terminal, it clears the screen and buffer. However, to the program, the operation places the device in a state from which the program can begin operations without leftover positioning from another program. The effect is the same, regardless of the device involved.

9.2 DEVICE DEPENDENT I/O

Device dependent I/O is supported on DX10 through I/O SVCs also. The function of the SVC itself determines whether it is device dependent or independent. For example, the Read Device Status SVC (>05) is a device dependent operation, since it performs differently depending on the device specified. It returns two bytes of status information from magnetic tape units, and four bytes of information from terminal devices. However, the call block is coded similarly to the call block for device independent I/O. The distinction lies with the *results* of the SVC operation.

Another type of device dependent I/O supported by DX10 is the extended control of terminal devices. Some SVC operations are device independent, unless you want to do some additional manipulation of a terminal device, such as sound the alarm, or control the cursor position. Then the operation becomes device dependent, since the operation is specifically for terminal devices.

9.3 PROGRAMMING CONSIDERATIONS

Most device independent I/O SVCs use the 12-byte call block, while the device dependent I/O SVCs use an extended call block. Often, the SVC call blocks for many operations are coded similarly, regardless of the results of the operation. (That is, regardless of whether the operation is device dependent or independent.) Some SVC operations for device dependent I/O require an extended call block to specify additional control parameters, but device independent I/O SVCs never require extended call blocks.

Table 9-1 lists all device I/O SVCs, and whether or not an extended call block is required for the operations. The label "Independent" means for device-independent I/O, "Dependent" means for device-dependent I/O, and "Terminal" refers to device-dependent I/O to terminal devices. "NE" means that the results are the same whether or not you use an extended call block. Generally, this happens when only a portion of the call block is read by the SVC. In such cases, you need not reinitialize the values in the extended call block, if you used one on the previous operation.

Table 9-1. Device I/O SVCs and Call Block Requirements

SVC Code*	Function	No Extended Call Block	Extended Call Block
>00	Open	Independent	Terminal
>01	Close	Independent	NE
>02	Close, Write EOF	Independent	NE
>03	Open and Rewind	Independent	NE
>04	Close and Unload	Dependent	Dependent
>05	Read Device Status	Independent	NE
>06	Forward Space	Independent	Dependent
>07	Backward Space	Independent	Dependent
>09	Read ASCII	Independent	Terminal
>0A	Read Direct	Dependent	Terminal
>0B	Write ASCII	Independent	Terminal
>0C	Write Direct	Dependent	Terminal
>0D	Write EOF	Independent	NE
>0E	Rewind	Independent	NE
>0F	Unload	Independent	Dependent

Independent — Refers to device independent I/O operations

Dependent — Refers to device dependent I/O operations

Terminal — Refers to device dependent terminal I/O operations

NE — Use of extended call block has no effect on the results

Note:

* SVCs >15 and >08 are not listed since their usage is restricted.

In general, a complete I/O transaction requires the following coding steps:

1. Assign a logical unit number (LUNO) to the device.
2. Open the logical unit for I/O.
3. Transfer data.
4. Close the logical unit.
5. Release the logical unit.

LUNOs and device access are discussed in detail in Section 3 of this manual. The remainder of this section describes the SVCs available on DX10 for performing Device I/O services.

Many SVCs described in this section can include an extended call block as explained earlier. A reference page illustrating all I/O SVCs and their call blocks and extended call blocks is located in Appendix E. In addition, the following paragraphs illustrate and explain both the general call block and extended call blocks required for device dependent I/O. When coding SVC calls from an assembly language program, use the following convention:

SVC @ADDRES

where ADDRES is the address of the call parameter block and SVC is a symbol for XOP 15.

NOTE

For all I/O, the SVC code is >00. In other words, the first byte of the SVC call block must contain zero. For the SVCs described in this section, the SVC subopcode given should be coded in byte 2 of the call block, unless otherwise stated. Refer to Appendix E for a complete I/O call block reference.

9.4 DEVICE I/O CALL BLOCKS

This paragraph describes the two call blocks available for device I/O SVCs. They are somewhat dissimilar, except that the SVC code required in byte zero is always zero to indicate I/O operations. One is required only for three particular SVCs. The second call block is generally used for all other device independent I/O SVCs.

The first call block is 27 bytes long, and could be called an "extended call block." This call block is required for the Assign LUNO SVC (subopcode >91), the Release LUNO SVC (subopcode >93), and the Verify Device Name SVC (subopcode >99), and is referred to as the Assign and Release LUNO SVC call block, and is shown in Figure 9-1. The byte and bit assignments are explained in Table 9-2. (This call block is quite similar to the call block used for file I/O, so it is also explained in Section 10).

Following the Assign and Release LUNO call block, Figure 9-2 and Table 9-3 show the usual I/O SVC call block and its bit assignments. The usual I/O call block applies to all other I/O SVCs, and can be extended for device dependent I/O as indicated in the figure.

Byte		
00	00	STATUS CODE
02	SUBOPCODE	LUNO
04	SYSTEM FLAG	USER FLAGS
06	DATA BUFFER ADDRESS	
08	LOGICAL RECORD LENGTH	
10	CHARACTER COUNT	
12	POINTER; SET TO ZERO FOR ASSIGN SVC	
14	RESERVED, SET TO ZERO INITIALLY	
16	UTILITY FLAGS	
18	RESERVED; SET TO ZERO	
20	RESERVED; SET TO ZERO	
22	DEVICE NAME ADDRESS POINTER	
24	RESERVED; SET TO ZERO	
26	RESERVED; SET TO ZERO	

2283137

Figure 9-1. Device I/O SVC Call Block for Assign and Release LUNO SVCs

NOTE

Some bytes used for file I/O SVCs are not used for device I/O, and must be initialized to zero as part of the call block coding for device I/O SVCs. All bytes marked RESERVED in the call blocks must also be initialized to zero.

Table 9-2. Device I/O Assign and Release LUNO Call Block Bit Assignments

Byte	Bit	Meaning
0		SVC code. This code is zero for all I/O operations.
1		Status code. This code is returned by the system (see the *DX10 Operating System Error Reporting and Recovery Manual*).
2		Utility subopcode. This byte is set by the user to specify the requested utility operation as follows: **Subopcode** — **Operation** >91 — Assign LUNO to pathname. >93 — Release LUNO assignment. >99 — Verify Device Name. All other utility subopcodes do not apply to device I/O.
3		LUNO. This byte specifies the LUNO to be used in the operation. It is supplied by the user except when the generate LUNO bit is set on an Assign operation. In this case, DX10 selects a LUNO and returns it in this field.
4–11		Not used. Reserved for compatibility with other calls.
12,13		Must be zero for device I/O.
14,15		Not used.
16		Utility flags. These flags are used for assign LUNO operations. For all other utility operations, set these flags to zero.
	0	Reserved. Set by the system, but does not apply to device I/O.
	1,2	Must be zero for device I/O.
	3,4	Scope of LUNO assignment. Set by the user to define the scope of the LUNO assignment or release.

Table 9-2. Device I/O Assign and Release LUNO Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
		Code — **Meaning** 00 — The LUNO is task local. 01 — The LUNO is station local. 10 — The LUNO is global. 11 — Reserved.
	5	Generate LUNO. Set by the user. If this flag is 1, DX10 automatically generates a LUNO number and returns it in the LUNO field (byte 3). The selection does not conflict with any LUNO accessible to the calling task. If this flag is a 0, the contents of the LUNO field is used. This flag applies to assign LUNO operations only.
	6,7	Must be zero for device I/O.
17–21		Must be zero for device I/O.
22,23		Device Name Pointer. These two bytes contain the user-specified address of a buffer that contains the device name used for assign LUNO operations. The device name format of the buffer is as follows: Byte 0 — Device name length (in bytes) = 4. Bytes 1-4 — Device name. Device names are assigned at system generation.
24–27		Must be zero.

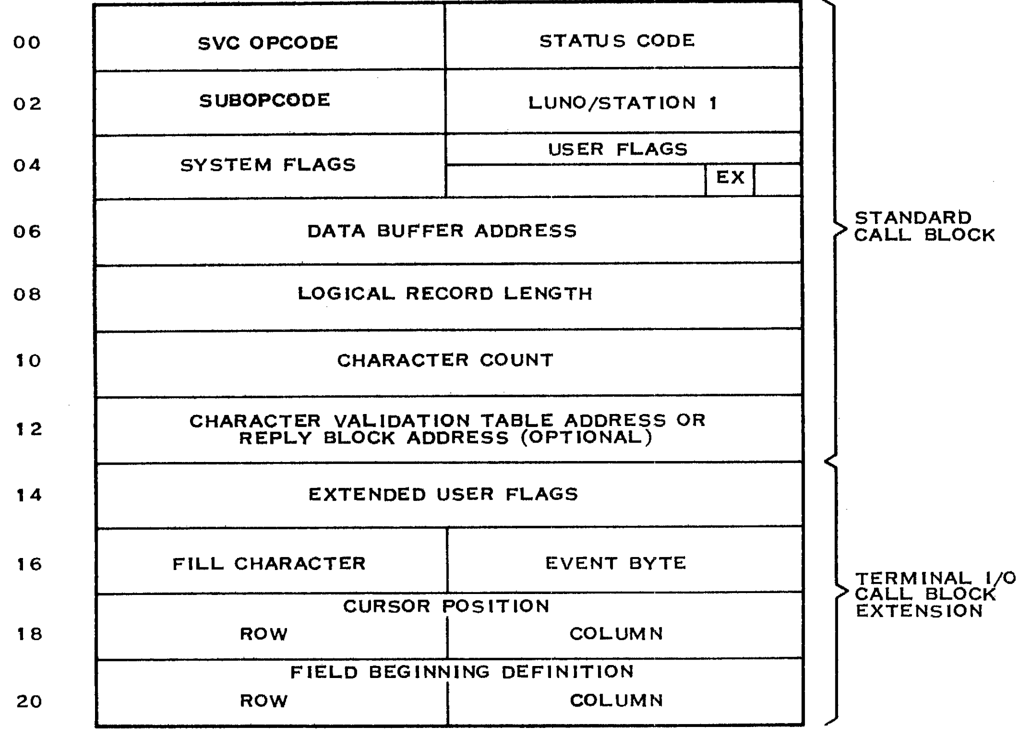


Figure 9-2. Device I/O SVC Call Block with Extended Block

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments

Byte	Bit	Meaning
0		Specifies SVC call type. Enter 00 to indicate I/O.
1		Used by the system to return status code (see the *DX10 Operating System Error Recovery Manual*).
2		Specifies SVC subopcode (see Table 9-4 for device response):
3		LUNO (Logical Unit Number).
4		System flags (set and reset by DX10).
	0	Busy. 1 = busy, 0 = done. Do not modify data buffer or call block when busy.
	1	Error. 1 = error, 0 = no error.
	2	EOF. 1 = end-of-file detected on input call, 0 = no EOF.
	3	Event character flag. Set to 1 if an input operation from a terminal was terminated by an even character. Event characters are described later in this section.
	4–7	Unused.
5		User flags. The user sets these flags to indicate the following to the system:
	0	Initiate. When this flag is set, the system returns control to the calling task without waiting for I/O completion.
	1	Output with Reply. See reply block (below). When set, this flag indicates two operations instead of one. For write operations, the reply flag indicates write followed by read. This flag applies only to write operations to terminal devices.

Byte 2 subopcodes:

Subopcode	Meaning
00	Open
01	Close
02	Close, Write EOF
03	Open and Rewind
04	Close and Unload
05	Read Device Status
06	Forward Space
07	Backward Space
08	Extended Device Control (device dependent)
09	Read ASCII
0A	Read Direct (device dependent)
0B	Write ASCII
0C	Write Direct (device dependent)
0D	Write EOF
0E	Rewind
0F	Unload

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
	2	Reserved — Set to zero.
	3	Reserved — Set to zero.
	4	Reserved — Set to zero.
	5	Immediate Open. If this bit is set to 1 on an Open or Open/Rewind for a teleprinter device, the open completes regardless of whether or not a terminal is connected to the port. If this bit is not set, the Open does not complete until the device is connected or a time-out expires.
	6	Use extended call block for terminal device.
	7	Blank Adjustment and Event Character Flag. This flag controls the blank adjustment option on READ or WRITE operations and the event character option on OPEN operations. Blank Adjustment: If this bit is set on read operations from CRU devices with variable records and the input logical record is shorter than the specified input buffer (bytes 8, 9) then the buffer is filled with blanks (>20) from the end of the input logical record to the end of the buffer. The value returned as the character count (bytes 10, 11) is equal to the adjusted record length including the blanks. In no case will the adjusted record length be greater than the system's device buffer size. On write operations to CRU devices with variable length logical records, all trailing blanks in the output buffer are deleted before the output is performed. The output character count specified in bytes 10 and 11 is not modified. Event Character Option: If this bit is set on an open operation to a terminal, then the event character option is selected for all subsequent read operations to that terminal. Event characters are described later in this section.
6,7		Data Buffer Address (must be even).
8,9		Logical Record Length. Set by the calling task on input operations to specify the maximum number of characters which may be stored in the input buffer. Not used for output operations. If this word is zero on an Open operation for any device except 979A tape, MT1600 tape, and disk, the system returns the maximum number of characters which may be transferred in a single I/O request.
10,11		Character Count. Set by the calling task on output to specify the number of characters to be output. Set by DX10 on input to specify the number of characters stored in the input buffer.

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
12,13		Reply Block Address. For output operations, if a reply is specified (byte 5, bit 1) then this word is the address of a three word block which contains the data buffer address, buffer length, and character count to be used for the reply input. These three words serve the same function for the reply input as bytes 6–11 do for read calls.
		Character Validation Table Address. If the Character Validation flag (bit 5 of byte 15) is set, this word is the address of the character validation table. Refer to the paragraphs on character validation later in this section.
Extended Call Block Assignments:		
		Note: Use this call block to specify special characteristics of a terminal. If you set byte 5, bit 6 to one, the system recognizes this portion of the call block. If byte 5, bit 6 is zero, the extended call block is not used. If you set byte 5, bit 6 to one, but this portion of the call block contains zeros, the system interpretes each flag and performs the function indicated for that flag when set to zero.
14	0	Field Start Specified Flag (all display terminals). Use with read and write calls in conjunction with bytes 20–21. Set this flag to one to indicate that the beginning of the field you want the system to read or write to is specified in bytes 20–21 of the call block. If this flag is zero, the current cursor position defines the start of the field. Specify the number of characters to read or write in bytes 8-9. You can control reading from the terminal buffer on a Read ASCII call by also using the Cursor Position flag (byte 15, bit 0), and specifying a cursor position in bytes 18–19.
	1	Intensity (all display terminals). This flag, when set, causes screen output to be in high intensity.
	2	Blink Cursor (all display terminals). This flag, when set, causes the cursor to blink on a read operation.
	3	Graphics (911, 931, 940, and Business System terminals). This flag, when set, displays control characters (characters >00 to >1F) as graphic characters on input and output. All characters are treated as data; no action is taken on control characters request. This flag is valid for read and write operations only. Also see the paragraphs on 8-Bit Code Support later in this section.
	4	8-Bit ASCII/Carriage Control (all terminals, both display and nondisplay). This flag is valid for Read Direct and Write Direct only. Bit 4 has several uses, as follows: • When set to one, bit 4 allows you to use bit 0 of each character to specify additional control for individual devices as follows: — For display terminals, bit 0 allows you to specify the intensity of each character of output on a Write Direct operation. — For Japanese and Arabic 911 VDTs, bit 0 identifies each character byte as Katakana or Arabic, respectively.

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
		• For teleprinter devices, bit 4 controls direct operations. If set to one on a Read Direct operation, the parity bit is left on the data instead of setting it to zero. If set to one on a Write Direct operation, the DSR parity setting is overridden and the parity bit remains the same as it is in the user's buffer. Also, the DSR does not check for event or special characters, and all data is passed directly through to the user's buffer. This flag also forces a carriage return and line feed at the end of a record on Read or Write ASCII calls for ASR and teleprinter devices.
		• If the CODE8 flag in the physical device table is set to one (during system generation) then the 8-bit ASCII carriage control is not supported. Refer to Appendix B for more information.
	5	Task Edit (all terminals, both display and nondisplay). This flag, when set, causes task edit characters entered on an input call to terminate the operation and to be returned in the Event Byte field of the call block. If the event key option was used for the open, but this flag is zero, task edit characters are ignored. Valid for Read ASCII and Write ASCII operations only. Task Edit keys are described later in this section.
	6	Beep (all terminals, both display and nondisplay). This flag, when set, produces an audible beep at the beginning of the input and at the end of the output. Valid for read and write operations only.
	7	Right Boundary (all display terminals). This flag, when set, limits character deletion or insertion to the present row when the field extends past the present row. Valid for read operations only.
15	0	Cursor Position flag (all display terminals). Used with Read Direct and Read ASCII calls to control reading from the terminal screen. You can use cursor positioning in conjunction with the Field Start flag and field start positioning. When used with Field Start positioning, you can read characters already displayed on the screen, beginning at a specified field start position, and ending with the character preceding the cursor. This allows the user to continue to input characters to the screen beginning at the cursor position, and not affect the field start position. When you set this flag, indicate the cursor position in bytes 18-19 of the extended call block. You control the number of characters read by coding the count into bytes 8 and 9 (logical record length bytes) of the general call block. At the completion of the Read ASCII operation, the current cursor position field is updated with the current cursor address. Cursor positions begin at row 0, column 0 (the upper left corner).
	1	Fill Character (all display terminals). This flag, when set, specifies that the character in byte 16 of the call block is to be used as a fill character whenever a character editing response needs a fill character. When this flag is not set, a blank will be used as the fill character. Valid for read operations only.

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
	2	Do Not Initialize Field (all display terminals). This user flag, when set to one, prevents the field from being initialized. When set to zero the input field is initialized with the fill character and the cursor is positioned to the start of the field. Valid for read operations only.
	3	Return on Termination Character (all display terminals). This flag, when set, requires a field termination character to be input before returning to the calling task. Once the input field is full, additional characters are accepted but not displayed or returned to the calling task. When the flag is clear, input terminates when a field termination character is entered or when the field is full. The cursor will advance one position past the end of a field if a field terminator is required. Field termination or error correction characters may be entered to the VDT. If the last position of the field is the last position of the row, the cursor is positioned to the next line. If the row is the last row of the screen, the cursor is turned off, giving the impression that it has advanced off the screen. Valid on read operations only. Valid for teleprinter devices on Read ASCII calls. For teleprinter devices on Read Direct calls with the eight-bit data option (byte 14, bit 4 set to 1), the read normally terminates when the DSR reads the specified number of characters or the time-out elapses. When the Return on Termination Character flag is set, the Read Direct call with the eight-bit data option also terminates when a character is encountered that is the same as the character in the event byte (byte 17) of the call block.
	4	Do Not Echo (all terminals, both display and nondisplay). This bit, when set, causes blanks to be sent to the screen rather than echoing the character input from the keyboard on Read ASCII operations. On Read Direct operations to teleprinters, this bit suppresses any character echo. Valid on read operations only.
	5	Character Validation (all terminals, both display and nondisplay). This flag, when set, specifies that DX10 will perform character validation on characters input to a field or record. A pointer to a validation table must be specified in bytes 12 and 13 of the supervisor call block. Character validation is not supported for the Write with Reply supervisor call. When this flag is not set, character validation is not performed. Valid on read operations only. Character validation and validation tables are discussed later in this section.

Table 9-3. General/Extended I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
	6	Field Begins in Error (all terminals, both display and nondisplay). This flag, when set, specifies that the supervisor call is reissued after an input field was entered and determined invalid. This flag is automatically reset by DX10 each time the supervisor call is executed. Valid on read operations only.
	7	Warning Beep (all terminals, both display and nondisplay). This flag, when set, causes an audible sound whenever functions are attempted such as left arrow when the cursor is at column zero, delete character when there are no characters to delete, or insert character when there is no room to insert characters. When this flag is not set, characters are allowed to be lost off the end of the field. Valid on read operations only.
16		Fill Character. Contains the ASCII representation of the character you wish to be used as a fill character. You must have bit 1 of byte 15 (user flags) set in order to specify a fill character.
17		Event Byte. This byte is used to return event key identification codes under certain circumstances. If bit 5 of byte 14 (user flags) is set, the task edit keys are also treated as event keys and are returned in this byte under certain circumstances. Event keys, system edit keys, and task edit keys are discussed later in this section.
18,19		Cursor Position. Use in conjunction with user flag bit 0 of byte 15 to specify the cursor position. Byte 18 indicates row, byte 19 indicates column. Used on a Read ASCII call in conjunction with Field Start positioning.
20,21		Field Start Position. Use to specify the beginning row and column position of the field you want to read from or write to. On a Read ASCII call, characters are read from the field beginning at the row and column specified by the Field Start field (bytes 20–21) in the call block, with the number of characters to read specified in the logical record length field (bytes 8–9). To control reading of characters within the specified field, set the Cursor Position flag (byte 15, bit 0) and specify the row and column in which to begin the read in the Cursor Position field (bytes 18–19). At the completion of the Read ASCII operation, the Cursor Position field is updated with the current cursor address. If the I/O call is not Read ASCII, the cursor is moved to the start of the field at the beginning of the I/O operation. The Cursor Position field is only used on Read ASCII I/O calls.

Notes:

Subopcode >08 is used to request device-dependent features that do not necessarily follow the standard description given. The specific parameters are determined by the device and the function being performed.

All Terminals designation includes teleprinter devices.

9.5 DEVICE DEPENDENT/INDEPENDENT I/O SVCS

Device independent SVCs are valid for all devices, except where the hardware limitations of the device make the operation invalid, as in a Close with EOF SVC (>02) operation to a card reader. Device dependent I/O operations may or may not be valid for particular devices. Table 9-4 shows the I/O SVC calls, and device responses to the calls.

Table 9-4. Device Dependent Responses to I/O Subopcodes

		I/O Operation															
		Open	Close	Close-EOF	Open-Rewind	Close-Unload	Read Status	Forward Space	Backward Space	Extended Device Control	Read ASCII	Read Direct	Write ASCII	Write Direct	Write EOF	Rewind	Unload
Device	**Opcode**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
733/743/745 ASR/KSR keyboard/printer		r	r	r	r	r	i	i	i	e	r	e	r	e	r	i	i
733 ASR cassette unit		r	r	r	r	r	i	r	r	e	r	r	r	r	r	r	r
Line printer		r	r	r	r	r	i	i	i	e	e	e	r	e	r	r	i
Card reader		r	r	e	r	r	i	i	i	e	r	r	e	e	e	i	i
Magnetic tape unit		r	r	r	r	r	r	r	r	e	r	r	r	r	r	r	r
Display terminal		r	r	r	r	r	r	i	i	e	r	r	r	r	i	r	i
Dummy device		r	i	i	r	i	i	i	i	e	r	r	r	r	r	r	i
820 KSR		r	r	r	r	r	r	i	i	e	r	r	r	r	r	r	i
Teleprinter devices		r	r	r	r	r	r	i	i	r	r	r	r	r	r	r	r

r — Device responds to opcode

i — Device ignores opcode

e — Device error code

The following SVCs can be used for either device dependent I/O, device independent I/O, or both. Those SVCs that can be used for both are explained as device independent first, and then as device dependent, giving the results of the call for each device.

Device dependent terminal I/O operations require extended call blocks. To code an extended call block, set bit 6 of byte 5 in the general call block to one, and then code the extended call block. (Byte 5 contains the user flags.) Refer to Figure 9-2 and Table 9-3 for a complete description of the extended call block that applies to many of the following SVCs.

9.5.1 >00 Subopcode — Open SVC

The Open operation should be performed before any data transfer operation. The LUNO remains open to the task until a Close operation (Close, Close with EOF, or Close Unload) is performed. The system returns a device type in bytes 6 and 7 of the call block and if the logical record length is zero (byte 8), a logical record length is returned in byte 8. The device types and default record lengths returned are given in Table 9-5

Table 9-5. Returned Device Types and Default Record Lengths

Device	Device Name 'xx' Indicates Number of Device	Device Type Returned	Record Length Returned
Dummy	DUMY	0000	0
820 KSR	ST08	0001	>A0
733 KSR	ST07	0001	>56
733 ASR	ST10	0001	>56
Teleprinter	ST09	0001	>200
Line Printer	LPxx	0002	>200
Cassette	CSxx	0003	>56
Card Reader	CRxx	0004	>A0
VDT	STxx	0005	>56
Disk	DSxx	0006	0
Diskette	DKxx	0006	>100
Communication	CMxx	0007	—
Mag Tape	MTxx	0008	0
Industrial	User Defined	000A	—
AMPL Emulator	EMxx	0003	0
AMPL Trace Emulator	TMxx	000F	0
File	User Defined	XXFF (XX is file type code)	LRL of file

For any file-oriented device, an Open operation (Open or Open Rewind) MUST be performed before any other operation. Consistent use of Open and Close is recommended for all files and devices, even though the Open call is not required for record-oriented devices.

9.5.1.1 Data Terminals. If the device specified is a data terminal, the Open Rewind operation clears the terminal's input buffer.

9.5.1.2 Printer Devices. If the device specified is a printer, the device performs a carriage return.

9.5.1.3 Video Display Terminals. If you do not use the extended call block, the cursor is positioned to column one of the bottom row of the screen.

9.5.2 >01 Subopcode — Close SVC

The Close operation informs the system that the current I/O sequence to the specified LUNO is complete. The LUNO may be subsequently reopened for further I/O. If a task terminates, DX10 automatically closes all LUNOs opened by the task. If the LUNO is task local the LUNO assignment is also deleted at task termination.

9.5.3 >02 Subopcode — Close with EOF SVC

The Close with EOF operation consists of a Write EOF operation followed by a Close Operation.

9.5.4 >03 Subopcode — Open Rewind SVC

The Open Rewind operation consists of an Open operation followed by a Rewind operation. It also clears the input buffer for terminal devices.

9.5.5 >04 Subopcode — Close Unload SVC

The Close Unload operation consists of an Unload operation followed by a Close operation.

9.5.6 >05 Subopcode — Read Device Status SVC

The Read Device Status operation provides the user access to device status information on certain devices. The user must specify the buffer address at which the returned status information is to be stored in bytes 6 and 7 of the call block and the length of the buffer in bytes 8 and 9 of the call block. The specified buffer must begin on a word boundary. The number of bytes returned by the Read Device Status operation will be indicated by bytes 10 and 11 of the call block. Bytes 10 and 11 will be zero if no information about the device is returned.

9.5.6.1 Magnetic Tape. The Read Device Status operation returns two bytes of status information as follows for magnetic tape units.

Byte	Code	Meaning
1	>80	Tape unit online, write ring installed.
	>40	Tape unit online, no write ring installed.
	>20	Tape unit offline.
2	>80	Phase-encoded recording, 1600 bits per inch (bpi).
	00	Nonreturn to zero inverted (NRZI) recording, 800 bpi.

9.5.6.2 Video Display Terminals. The Read Device Status operation returns four bytes of status information for VDTs. The first byte will contain the maximum row address of the VDT and the second byte will contain the maximum column address. The third and fourth bytes contain the number of keyboard input characters currently internally buffered in the character queue. This information is returned in bytes 0 through 3 for the 911, 931, 940, and Business System terminals.

The 931, 940, and Business System terminals return 34 additional bytes of information, provided the buffer has sufficient space.

Byte	Bit	Meaning
0		Maximum row address (24)
1		Maximum column address (80)
2,3		Number of keyboard characters in KSB input queue
4		DSR type (>31 indicates 931, 940, or Business System; >40 indicates 940 or Business System before Release 3.6)
5		Port identification number
6,7		CRU/TILINE address of hardware interface
8,9		Reserved (>FFFF)
10		Hardware interface type >01 = CI401 >06 = Business System 300 EVDT port >07 = 990/10A 9902 port >08 = CI402 9902 port >09 = CI421 9902 port >0A = CI422 9902 port >23 = CI403 port >24 = CI404 port >30 = CI421 9903 port
11		Reserved (>00)
12,13		Edit flag word 1
14,15		Edit flag word 2
16,17		Reserved (>0000)
18,19		Reserved
20		Reserved (>00)
21		Reserved (>00)
22		Line flags
	0	Half duplex
	1	Switched line (defined during system generation)
	2–7	Reserved (0)
23		Reserved (>00)

Byte	Bit	Meaning
24		Speed code 0 = 50 baud 1 = 75 baud 2 = 110 baud 3 = 134.5 baud 4 = 150 baud 5 = 200 baud 6 = 300 baud 7 = 600 baud 8 = 1,200 baud 9 = 1,800 baud A = 2,400 baud B = 3,600 baud C = 4,800 baud D = 7,200 baud E = 9,600 baud F = 14,400 baud 10 = 19,200 baud 11 = 28,800 baud 12 = 38,400 baud
25-35		Reserved (>00)
36		Terminal type code (>A1 = 931, >B0 = 940)
37		Reserved (>00)

9.5.6.3 Disk Devices. The Read Device Status operation returns 10 bytes of status information for disk units.

Byte	Bit	Meaning
0,1		Number of words per track
2		Number of sectors per track
3		Number of overhead words per record
4,5	0-4	Number of heads
	5-15	Number of cylinders
6		Number of sectors per record
7		Number of records per track
8,9		Number of words per record

9.5.6.4 Teleprinter Devices. Read Device Status for teleprinter devices returns as many as 58 status bytes when the specified buffer is large enough. The location and size of the buffer are specified in the call block. The buffer has the following format:

Byte	Bit	Contents
0,1		Reserved (>FFFF)
2,3		Number of keyboard characters in KSB input queue
4		DSR type (always >05 for TPD)
5		Reserved (>00)
6,7		CRU/TILINE address of hardware interface
8,9		CRU address of associated automatic call unit
10		Hardware interface type >01 = CI401 >05 = TTY/EIA >06 = a 9902 port
11		Reserved (>00)
12,13		Read ASCII time-out in .25-second counts
14,15		Write ASCII and direct time-out in .25-second counts
16,17		Read direct time-out in .25-second counts
18,19		Read direct time-out for characters 2–n
20		Reserved (>00)
21		State flags
	0	Online
	1	Connect in progress
	2	Open
	3	DLE received
	4	Half-duplex line belongs to remote terminal
	5	Resend
	6	Spare
	7	Eight-bit data queuing
22		Line flags
	0	Half-duplex
	1	Switched line
	2	Refuse call
	3	Automatic disconnect enabled
	4	DLE/EOT for disconnect sequence
	5	Reserved
	6	Exclusive access
	7	Half-duplex LTA enabled
23		Access flags
	0	No echo
	1	Spare
	2	Transmit parity enabled
	3,4	Transmit parity type
	5	Receive parity enabled
	6,7	Receive parity type

Byte	Bit	Contents
24		Speed code 0 = 110 baud 1 = 300 baud 2 = 600 baud 3 = 1200 baud 4 = 2400 baud 5 = 4800 baud 6 = 9600 baud − 1 = 300 or 1200 as selected by the modem when using 212 type modems
25		End-of-record character
26		End-of-file character
27		Line turnaround character
28		Parity error substitute character
29		Carriage return delay count (.25-second interval)
30–33		Reserved (>00)
34,35		Maximum characters buffered
36		Terminal type code (hexadecimal) 3 = 703 51 = 781 7 = 707 53 = 783 2B = 743 55 = 785 2D = 745 57 = 787 3F = 763 78 = 820 41 = 765 8C = 840
37		Last character received
38–40		Reserved
41		Speed specified during system generation
42,43		Reserved
44,45		Time-out specified during system generation
46,47		Number of parity errors detected
48,49		Number of lost characters
50,51		Number of reads
52,53		Number of writes
54,55		Number of other I/O calls
56		Number of retries
57		Number of LUNOs assigned

9.5.6.5 Line Printers. The Read Device Status operation returns 38 bytes of status information for line printers provided the buffer has sufficient space. The buffer format differs for printers that use asynchronous hardware interfaces and printers that use the CI403 and CI404 boards. The buffer has the following format for printers that use asynchronous hardware interfaces:

Byte	Bit	Contents
0		Reserved (>00)
1		Reserved (>FF)
2,3		Reserved (>0000)
4		DSR type (>01 indicates any printer except one that uses a CI403 or CI404 board; >31 indicates an attached printer)
5		Reserved (>00)
6,7		CRU/TILINE address of hardware interface
8,9		Reserved (>FFFF)
10		Hardware interface type
		>05 = TTY/EIA
		>06 = a 9902 port
		>80 = parallel printer
11-21		Reserved (>00)
22		Line flags
	0	Half duplex
	1	Switched line (defined during system generation)
	2-7	Reserved (0)
23		Reserved (>00)
24		Speed code
		0 = 50 baud
		1 = 75 baud
		2 = 110 baud
		3 = 134.5 baud
		4 = 150 baud
		5 = 200 baud
		6 = 300 baud
		7 = 600 baud
		8 = 1,200 baud
		9 = 1,800 baud
		A = 2,400 baud
		B = 3,600 baud
		C = 4,800 baud
		D = 7,200 baud
		E = 9,600 baud
		F = 14,400 baud
		10 = 19,200 baud
		11 = 28,800 baud
		12 = 38,400 baud
25-37		Reserved (>00)

The buffer has the following format for printers that use a CI403 or CI404 board:

Byte	Bit	Contents
0		Reserved (>00)
1		Reserved (>FF)
2,3		Reserved (>0000)
4		DSR type (>10 indicates a printer that uses a CI403 or CI404 board)
5		Port identification number
6,7		CRU/TILINE address of hardware interface
8,9		Reserved (>FFFF)
10		Hardware interface type
		>23 = CI403 port
		>24 = CI404 port
11-21		Reserved (>00)
22		Line flags
	0	Half duplex
	1	Switched line (defined during system generation)
	2-7	Reserved (0)
23		Reserved (>00)
24		Speed code
		0 = 50 baud
		1 = 75 baud
		2 = 110 baud
		3 = 134.5 baud
		4 = 150 baud
		5 = 200 baud
		6 = 300 baud
		7 = 600 baud
		8 = 1,200 baud
		9 = 1,800 baud
		A = 2,400 baud
		B = 3,600 baud
		C = 4,800 baud
		D = 7,200 baud
		E = 9,600 baud
		F = 14,400 baud
		10 = 19,200 baud
		11 = 28,800 baud
		12 = 38,400 baud
25-37		Reserved (>00)

9.5.7 >06 Subopcode — Forward Space SVC

The Forward Space operation applies only to the 733 ASR cassette units and the magnetic tape units, and is ignored by all other devices. The number of logical records to be moved is placed in bytes 10 and 11 of the call block, and the Forward Space operation moves forward the specified number of logical records or until an end-of-file (EOF) record is encountered. If an EOF record is not encountered, zero is returned in bytes 10 and 11 of the call block. If an EOF record is encountered, the tape is positioned following the EOF record, the EOF flag (byte 4, bit 2 of the call block) is set, and the number of records remaining is returned in bytes 10 and 11 of the call block.

9.5.8 >07 Subopcode — Backward Space SVC

The Backward Space operation applies only to the 733 ASR cassette units and the magnetic tape units, and is ignored by all other devices. The number of logical records to be moved is placed in bytes 10 and 11 of the call block, and the Backward Space operation moves in the reverse direction the specified number of logical records or until an end-of-file (EOF) record is encountered. If an EOF record is not encountered, zero is returned in bytes 10 and 11 of the call block. If an EOF record is encountered, the tape is positioned so the next read operation encounters the EOF record, the EOF flag (byte 4, bit 2 of the call block) is set, and the number of records remaining is returned in bytes 10 and 11 of the call block.

9.5.9 >09 Subopcode — Read ASCII SVC

The Read ASCII operation reads a record from the selected device and stores the characters in the specified buffer, two characters per word. The buffer address is specified in call block bytes 6 and 7 and the buffer size is specified in bytes 8 and 9. The actual number of characters stored is returned in the character count, bytes 10 and 11. If the length of the input record exceeds the buffer size, the remaining characters are not available.

The ASCII characters listed for each device in Appendix B are valid and are stored with the most significant bit set to zero. All other characters are ignored, except as noted for the card reader.

9.5.9.1 Magnetic Tape Unit. When the device specified is a magnetic tape unit, the Read ASCII operation reads the current record and transfers the available characters to the specified buffer in memory.

9.5.9.2 Video Display Terminal. Characters are transferred to the buffer until the buffer is full, an end-of-record character is read, an event character is read, an end-of-file character is read, or a task edit character is read. Characters can be corrected by pressing the Previous Character key and then entering the correct character. EOF is the Enter key. If you do not use the extended call block, all characters entered are displayed at the current cursor position in low intensity. The Return key (>0D) positions the cursor at column one of the current row.

9.5.9.3 Teleprinter Devices. Characters are transferred to the buffer until the buffer is full, an end-of-record character is read, an event character is read, an end-of-file character is read, or a task edit character is read. You can correct typing errors by pressing CTRL and H simultaneously. The teleprinter performs a backspace and a line feed operation on the initial CTRL/H and just a backspace on each subsequent CTRL/H. The effect of a backspace on a character is to delete that character; when you backspace to a character and correct it, all the following characters must be reentered. The end-of-record and end-of-file characters are user-modifiable parameters of the teleprinter device DSR. The default values are CR (>0D) for the end-of-record and EM (>19) for end-of-file. If using half- duplex, the line turnaround character is interpreted as an end-of-record character. Devices specified as types ASR, KSR, or 820 during system generation use CR as end-of-record and DC3 (>13) as end-of-file.

9.5.9.4 733 ASR Cassette Unit. When the device specifed is a 733 ASR cassette unit, the Read ASCII operation transfers characters from the cassette to the specifed buffer until a carriage return is detected or the buffer is full. The maximum number of characters in a cassette record is 86. An end-of-file record on a cassette includes a DC3 (X-OFF) as the first character. The cassette unit does not provide a logical end-of-medium indication, but does provide a physical end-of-tape indication which may occur at either end of the tape.

9.5.9.5 Card Reader. When the device specified is a card reader, the Read ASCII operation reads a card and transfers the characters read to the specified buffer. Up to 80 characters are transferred. When more than 80 characters are specified, only the first 80 are read and transferred. When fewer than 80 characters are specified, the remaining characters on the card are not read and transferred. Characters other than standard 990 characters for the card reader (>20 through >50) are placed in the buffer as spaces, and the system returns an error status code when these characters are read. The end-of-file record for the card reader has a slash in the first column and an asterisk in the second column (/*).

9.5.9.6 Other Devices. The Read ASCII operation is an illegal operation for output-only devices like the line printer, and the system returns an error status (see *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI)).

9.5.10 >0A Subopcode — Read Direct SVC

The Read Direct operation reads the number of characters you specify, and transfers the characters to the buffer at the address you specify. The characters are transferred just as received from the device, without being translated by DX10's device service routine for that device. The number of characters actually read is returned in bytes 10 and 11 of the call block. This SVC is always device-dependent since the results are different depending on the device.

9.5.10.1 733 ASR Cassette Unit. When the device specified is the cassette unit, the system sets the most significant bit of each character to zero and stores two characters per word. The read operation terminates when a carriage return is encountered, or when the complete record is read.

9.5.10.2 Magnetic Tape Unit. When the device specified is a magnetic tape unit, the Read Direct operation is the same as the Read ASCII (subopcode >09) operation.

9.5.10.3 Card Reader. When the device specified is the card reader, a column on the card is stored in a word of the buffer. The four most significant bits of the word are set to zero. The rows of the card are represented in bits 4–15 as follows: if a hole is punched, the corresponding bit in the word contains a one; if a hole is not punched, the corresponding bit contains a zero. Bits 4–15 correspond to rows 12, 11, then 0–9 respectively, that is, from top to bottom of a column. The entire record is transferred to the buffer. End-of-file is undefined for a Read Direct operation.

9.5.10.4 Video Display Terminal. When the specified device is a display terminal, the specified number of characters is read from the screen (not the keyboard) starting at the cursor position. If you do not use the extended call block, all characters entered are displayed at the current cursor position in low intensity. Pressing the Return key (>0D) positions the cursor at column one of the current row.

9.5.10.5 820 KSR. When the device specified is an 820 KSR, the Read Direct operation reads characters from the device until the buffer is full or a carriage return is detected. No translation of characters is performed.

9.5.10.6 Teleprinter Devices. When the device specified is a teleprinter device, the Read Direct operation reads a record from the device. The operation is terminated upon receipt of the record terminator, the EOF character, or expiration of the read time-out. The data is stored in a specified buffer, two characters per word. The parity bit of each character is set to zero unless eight-bit data is specified.

If eight-bit data is specified, no translation of characters is performed. That is, no parity checking or event characters occur. The operation is terminated upon receipt of the number of characters specified or expiration of the read time-out. The record terminator and the EOF character are not recognized.

9.5.10.7 Other Devices. When a Read Direct operation is attempted on other devices, the system returns an error status code (see the *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI)).

9.5.11 >0B Subopcode — Write ASCII SVC

The Write ASCII operation writes a record consisting of ASCII characters to the specified device from the buffer at the address you specify in the call block. The buffer contains two characters per word. You specify the number of characters to be written in bytes 10 and 11 of the call block. The device service routine for each device recognizes the characters, and sends it to the device in the appropriate form. Invalid characters can produce erratic results. Refer to Appendix B for a list of the character codes recognized by each terminal device service routine.

9.5.11.1 Magnetic Tape. When the device specified is a magnetic tape unit, the Write ASCII operation transfers data from the specified user buffer and writes the available characters to tape.

9.5.11.2 Keyboard/Printer. When the device specified is the keyboard printer, the characters are printed. An HT (>09, Horizontal Tab) character results in a space, and FF (>0C, Form Feed) character results in eight line feed operations.

9.5.11.3 Line Printer. When the device specified is the line printer, all characters, including control characters, are sent to the printer. The printer response depends upon the type of printer used and the installed options. For details, refer to the appropriate installation and operation manual for the printer being used.

9.5.11.4 Video Display Terminal. When the device specified is a display terminal, the characters are displayed on the screen. If you do not use the extended call block, the output is displayed at the current cursor position in low intensity. Pressing the Return key (>0D) positions the cursor at column one of the current row. A line feed (>0A) moves the contents of the display up one line and positions the cursor at column one of the current row.

9.5.11.5 733 ASR Cassette Unit. When the device specified is a 733 ASR Cassette Unit, the ASCII characters are written on the cassette. A carriage return in the buffer is translated to an End Transmit Block (ETB, >17) character. The maximum number of characters to be written is 86. The system then writes a carriage return, a line feed, a DC4 (Record OFF) character, and a DEL (Rubout) character. The end-of-file record contains a DC3 (X-OFF) character, a DC4 (Record OFF) character, and a DEL (Rubout) character. When the Reply bit is set, the response is read from the other cassette unit.

9.5.11.6 820 KSR. When the device specified is an 820 KSR, the characters are printed on the terminal. An HT (>09, Horizontal Tab) character results in a space.

9.5.11.7 Teleprinter Devices. When the device specified is a teleprinter device, the Write ASCII operation prints the specified record on the terminal. The location of the record and the number of characters output are specified in the call block. The parity bit for each character is set as defined by the configuration flags.

9.5.11.8 Other Devices. When a Write ASCII operation is attempted on an input only device such as the card reader, the system returns an error status code (see *DX10 Operating System Error Recovery and Reporting Manual* (Volume VI)).

9.5.12 >0C Subopcode — Write Direct SVC

The Write Direct operation writes a record without performing any translation and writes the characters from the buffer at the address in bytes 6 and 7 of the call block. The number of characters to be written is specified in bytes 10 and 11 of the call block.

9.5.12.1 733 ASR Cassette Unit. When the device specified is the 733 ASR Cassette Unit, the seven least significant bits of each byte are written on the cassette. The maximum number of characters per record is 86. If a DC4 (Record OFF) character is embedded in the buffer, the system writes the DC4 character and automatically writes a DC2 (Record ON) character to continue the operation. To assure the last record is actually written on the tape, the user task places a carriage return in the buffer. When the reply bit is set, the response is read from the other cassette unit.

9.5.12.2 Magnetic Tape Unit. When the device specified is a magnetic tape unit, the Write Direct operation is the same as the Write ASCII (subopcode >0B) operation.

9.5.12.3 Video Display Terminal. When the specified device is a display terminal, characters appear on the screen. If you do not use the extended call block, output is displayed at the current cursor position in low intensity. A carriage return (>0D) positions the cursor at column one of the current row. A line feed (>0A) moves the contents of the display up one line and positions the cursor at column one of the current row. If the CODE8 flag in the physical device table is set to one (during system generation), the 8-bit ASCII flag (byte 14, bit 4) is not supported on a Write Direct operation.

9.5.12.4 820 KSR. When the device specified is an 820 KSR, the Write Direct command outputs the characters in the buffer directly with no processing of event characters or control characters.

9.5.12.5 Teleprinter Devices. When the device specified is a teleprinter device, the Write Direct command outputs the characters in the buffer directly.

9.5.12.6 Line Printer. When the device specified is the line printer, all characters including control characters are sent to the printer. The printer response depends on the type of printer used and its attached options. For details, refer to the appropriate installation and operation manual for the printer being used.

9.5.12.7 Other Devices. When the Write Direct operation is attempted on any other device, the system returns an error status code (see the *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI)).

9.5.13 >0D Subopcode — Write EOF SVC

The Write EOF operation consists of writing the end-of-file record defined for the specified device.

9.5.13.1 733 ASR Cassette Unit. When the device specified is a 733 ASR Cassette Unit, the operation writes a DC3 (X-OFF) character on the cassette.

9.5.13.2 Keyboard/Printer. When the device is a keyboard/printer, the operation performs three line feed operations.

9.5.13.3 Line Printer. When the device is a line printer, the operation performs a form feed operation.

9.5.13.4 Video Display Terminal. When the device is a VDT, the operation is ignored.

9.5.13.5 Card Reader. When the device is the card reader, the system returns an error status code, and no operation is performed.

9.5.13.6 Magnetic Tape Unit. When the device is a magnetic tape unit, the operation writes the magnetic tape EOF mark.

9.5.13.7 820 KSR. When the device specified is an 820 KSR, the Write EOF command performs a form feed.

9.5.13.8 Teleprinter Devices. When the device specified is a teleprinter, the Write EOF command performs a form feed or three line feeds, depending on the presence of form control on the attached terminal.

9.5.14 >0E Subopcode — Rewind SVC

The Rewind operation is specified by I/O operation code >0E. When the specified device is the cassette unit, the operation rewinds the cassette tape to the clear area at the beginning of the tape, and then moves the tape to the beginning of the tape, illuminating the READY indicator on the 733 ASR terminal. When the specified device is the line printer, the operation performs a form feed operation. When the device is a magnetic tape unit, the operation rewinds the tape to the load point and places the unit in the ready state. When the device is a VDT, the screen is erased. The Rewind operation clears the input character queue for teleprinter devices. The Rewind operation is ignored by other devices.

9.5.15 >0F Subopcode — Unload SVC

The Unload operation applies to the cassette unit, the magnetic tape unit, and teleprinter devices. The Unload operation for the cassette unit consists of rewinding the cassette tape to the clear area at the beginning. For the magnetic tape unit, the operation consists of rewinding the tape to the physical beginning in preparation for unloading the reel. For teleprinter devices, the Unload operation disconnects terminals attached through switched circuits.

9.5.16 >91 Subopcode — Assign LUNO SVC

The Assign LUNO SVC for devices uses an extended call block similar to that required for file I/O. The Assign LUNO SVC is discussed in paragraph 9.4, along with the description of the required call block. The call block described also applies to the Release LUNO SVC (>93) and the Verify Device Name SVC (>99).

9.5.17 >93 Subopcode — Release LUNO SVC

The Release LUNO SVC for devices is discussed in paragraph 9.4, along with the required call block. The call block described also applies to the Assign LUNO SVC (>91) and the Verify Device Name SVC (>99).

9.5.18 >99 Subopcode — Verify Device Name SVC

The Verify Device Name SVC is discussed in paragraph 9.4, along with the required call block. The call block described also applies to the Assign LUNO (>91) and Release LUNO (>93) SVCs.

9.6 KEY TYPES

DX10 separates the keys on a keyboard into four types:

- Data keys — Alphanumeric and special characters only; not programmable.
- Event keys — Programmable.
- Task edit keys — Programmable, but also have a special function.
- System edit keys — Not programmable; the DSR always recognizes these keys and performs a special function in response.

Each of these key types is discussed in the following paragraphs.

NOTE

> For code compatibility, the system and task edit key characters are transformed from ASCII representation to a DSR output code. Appendix B contains a list of DSR output code transformations.

9.6.1 Data Keys

Data keys include the printable characters and do not include the control characters (ASCII 00 - 1F and 7F). When data keys are struck, the corresponding ASCII character code is placed in the data buffer. You cannot alter the returned keycode without modifying the DSR.

9.6.2 Event Keys

Event keys usually include the keys F1 through F14, Command, and Print.

The DSR handles event characters along with data and edit keys in chronological order. If the open with event key option is selected, pressing an event key terminates input to the buffer and causes an immediate return to the task that executed the Read SVC. If you press an event key while no input is in progress, the next executed Read SVC to the terminal is terminated by the event character. No data is lost provided the character queue is sufficient to capture keystrokes that the operator types ahead of the program. If you do not open with event key, then an event key does not terminate I/O, and the event character is discarded.

Event keys are programmable. This means that the keycode from the DSR is available to user tasks. Any function desired by pressing these keys must be implemented in the user task. DX10 does nothing with these keys. There are two ways to access event characters:

- You can use the Get Event Character supervisor call (SVC) if no input is in progress, or if the I/O call in use is a device-independent call. The Get Event Character SVC can access a device character buffer by LUNO (SVC code >30) or by station ID (SVC code >39).

- You can examine the contents of byte 17 of the extension block if an extended input call is in progress. The event character is returned in byte 17. The keyboard must be opened with the event character option specified.

The Get Event Character SVCs are described later in this section along with other I/O related SVCs. You specify the Event Key option using the Open SVC and setting the event key flag (byte 5 bit 7 of the call block) to one. The task can then obtain event keys from the input buffer using the Get Event Key SVC or with a read call using an extended call block. (The extended call block format is described earlier in this section.)

With the Event Key option, event keys act as field terminators on read calls. The event key flag in the system flags field (byte 4 bit 3 of the call block) is set to one when an event key terminates a read operation. The event key is returned in byte 17 if an extended call block was specified; otherwise the user must issue a Get Event Key SVC to obtain the event key.

CAUTION

Any task may access the character input buffer by performing I/O to the same LUNO. Therefore, programs that attempt to share the terminal with other programs might fail if another task removes the event key from the character buffer. The user must be especially careful when the task is performing I/O to a device to which SCI is also performing I/O. Both the user task and SCI may have a confused response.

The event keys which may be programmed from the task level are the following:

Key	Code
F1	>81
F2	>82
F3	>83
F4	>84
F5	>85
F6	>86
F7	>96
F8	>97
F9	>80
F10	>9A
F11	>9C
F12	>9D
F13	>9E
F14	>9F
Command	>98
Print	>99

9.6.3 Task Edit Keys

Task edit keys usually include Erase Input, Home, Skip, Forward Tab, Previous Field, Previous Line, Next Line, Next Field, Enter, Initialize Input, and Return. These keys generate codes returned to the calling task under the following conditions:

- If no input from the terminal is in progress, or if standard device independent I/O is in progress, no code is returned except that the Return and Enter keys constitute end-of-record and end-of-file characters, respectively.

- If an input operation using the extended call block is in progress, and if bit 5 of byte 14 is set to one, these characters are treated as event characters and returned in byte 17 of the extended call block.

Several of the indicated keys are both task edit keys and system edit keys. The following paragraphs discuss the effects of each key. The character code transformation tables in Appendix B indicate the key types and contain further information about the keys and associated codes output by the terminal DSR. The task edit keys are as follows:

- The Erase Input key functions the same as the Erase Field key but returns to the calling task with the character code >8E in byte 17 of the extended call block. (This key is also a system edit key.)

- The Home key positions the cursor at the beginning of the field and returns to the calling task with the character code >8C in byte 17 of the extended call block. (This key is also a system edit key.)

- The Skip key fills the field from the present cursor position to the end of the field with the fill character, places the cursor one character position beyond the field, and returns to the calling task with character code >8B in byte 17 of the extended call block. (This key is also a system edit key.)

- The Forward Tab key accepts all characters from the start of the field cursor position to the first fill character (if other than a blank) or to the end of the field, leaves the cursor at the position where the tab occurred, and returns to the calling task with the character code >89 in byte 17 of the extended call block. (This key is also a system edit key.)

- The Previous Field key positions the cursor at the beginning of the field and returns to the calling task with the character code >94 in byte 17 of the extended call block.

- The Previous Line key returns to the calling task with the character code >95 in byte 17 of the extended call block.

- The Next Line key returns to the calling task with the character code >8A in byte 17 of the extended call block.

- The Next Field key functions the same as the Forward Tab key except that the cursor is placed one character position past the end of the field and the character code is >87.

- The Enter key functions the same as the Forward Tab key except that the cursor is placed one character position past the end of the field, and it sets the EOF bit. The character code is >93.

- The Initialize Input key creates a blank line, and positions the cursor at the beginning of that blank line. The character code is >8F.

- The Return key indicates end-of-record for input, and moves the cursor to the beginning of the next line. The character code is >8D. (This key is also a system edit key.)

9.6.4 System Edit Keys

System edit keys generally have special meaning to DX10, and are not programmable. That is, the code is not available to user tasks. For those system edit keys that are also task edit keys, the DSR always recognizes the character and performs a specific function, but the character is also available to user tasks. The character code transformation tables in Appendix B indicate which keys are task edit and system edit keys.

The system edit key functions are as follows:

- The Erase Field key fills the field with the fill character and positions the cursor at the beginning of the field.

- The Delete Character key deletes the input character under the cursor, moves all characters within the field to the right of the cursor left one character position, replaces the last character position of the field with the designated fill character, and leaves the cursor at its position prior to the keystroke. If there are no characters from the present cursor position to the right end of the field, this key gives a warning beep, if the warning beep bit is set.

- The Insert Character key gives a warning beep if the character under the cursor is the fill character and not a blank. Otherwise, it sets the input mode to insert characters, and gives a warning beep when an inserted character causes characters to be lost on the right end of the field (if the warning beep bit is set), but lets the characters be lost off the end of the line if the warning beep bit is not set (any keystroke not entering data will change the input mode back to the noninsert mode).

- The Previous Character key gives a warning beep if the cursor is at the beginning of the field and the warning beep flag is set in the extended call block. Otherwise, it moves the cursor left one character position.

- The Next Character key leaves the character under the cursor intact, moves the cursor right one character position, and, if the cursor moves beyond the end of the field, returns to the calling task (if designated by the extended call). This key gives a warning beep on keystrokes which would cause the cursor to move beyond the end of the field (if the warning beep flag is set).

- The Return key serves as an end-of-record for input and moves the cursor to the beginning of the next field.

- The Attention key implements a wait function. The DSR waits for the next character typed, and interprets it as follows:

  — Return — Causes an I/O SVC error code >06 (timeout) to be returned to the task that did the read or write operation, and the output is aborted.

  — Control/X — Aborts the task that made the I/O call or the most recently executed task associated with the terminal.

  — ! — Activates SCI if not already active at the station, and otherwise behaves as any other character.

  — Any other (including the Attention key again) — Resumes output normally. You can use the Attention key to halt and resume listing displays produced by an LD command, for example.

9.6.5 Repeat Character Compression

The repeat character compression option is supported on all VDTs. This option saves space by enabling a string of repeated characters, such as a line of underscores, to be represented by only three characters. These characters consist of the character to be repeated, a replication character of > 7F, and a repeat count. A repeat count of zero displays the character to be repeated followed by > 7F; a repeat count of "n" displays the character to be repeated "n" number of times. For example, > 20, > 7F, > 06 results in six blanks (> 20) being displayed. The character sequence > 20, > 7F, > 00 results in one blank plus > 7F being displayed. Repeat character compression transmits all character codes to the terminal; that is, if the repeat character is a line feed, carriage return, or form feed, the system displays the graphic equivalent of that character for the associated terminal.

When using repeat character compression, a value of three must be specified in the character count field (bytes 10–11) of the I/O call block.

9.6.6 Character Validation

DX10 supports character validation for ASCII input from terminals. Character validation is specified in the Read ASCII SVC. A validation table is supplied for each read with validation call.

To specify character validation, an extended call block must be used, with the Validation Field flag (byte 15 bit 5 of the call block) must be set to one. The table address must be specified in bytes 12 and 13 of the extended call block. The Validation Field flag indicates to the DSR that character validation is to be performed. Character validation is not supported on Write with Reply SVCs.

The character validation table has the following format:

Byte	Bit	Meaning
0		Number of bytes (including this one) in the validation table, equal to ((number of ranges) × 2) + 2.
1		Flags.
	0	Set to zero, this flag indicates that the character ranges specified in the table represent valid characters to be accepted upon input. Set to one, this flag indicates that the character ranges specified in the table represent invalid characters to be rejected on input.
	1-7	Reserved.
2		Lower character code in range 1 (inclusive).
3		Upper character code in range 1 (inclusive).

Repeat byte pairs for character ranges 2 through n.

Characters entered for validation are echoed, the cursor is advanced, and then validation is performed. If validation fails, the field enters character error mode. In this state, any character other than an error correction character will:

- Beep (if the Warning Beep flag is set)
- Not echo
- Not advance the cursor. Error correction keys are space bar, Next Character, Previous Character, Next Field, Previous Field, Erase Field, and Erase Input.

NOTE

When the Carriage Control flag (byte 14 bit 5 of the call block) is set, the Next Field and Previous Field keys will appear to be inoperative. If this occurs, use another error correction key.

Validation can also be performed on the field level by the user task. When the field entry is tested by the task, and the entry is in error, the I/O call can be reissued with the Field Begins in Error (bit 6 byte 15 of the call block) set. This call repositions the cursor at the beginning of the field to allow the user to reenter the data in the field. Character validation is also performed on subsequent data entries. The DSR resets the Field Begins in Error flag when the new data is entered.

An example of the character validation is as follows: The valid inputs to a field are the numerals 0-9 and the uppercase letters A-Z. The validation table can be constructed as shown in the following example:

```
         EVEN
TABLE    BYTE   6        NUMBER OF BYTES IN TABLE
         BYTE   0        TABLE CONTAINS VALID ENTRIES
*                        (FLAG SET TO ZERO)
         BYTE   >30      LOWER  CODE  IN RANGE  1-0
         BYTE   >39      UPPER  CODE  IN RANGE  1-9
         BYTE   >41      LOWER  CODE  IN RANGE  2-A
         BYTE   >5A      UPPER  CODE  IN RANGE  2-Z
```

An example of the SVC call block is as follows:

```
RASCI    DATA   0        I/O OPCODE AND STATUS CODE
         BYTE   >09      OPCODE FOR READ ASCII SVC
         BYTE   >2B      LUNO IS >2B
         BYTE   >0       SYSTEM FLAGS
FLAG1    BYTE   >42      SET OUTPUT WITH REPLY AND
*                        EXTENDED CALL BLOCK USER FLAGS
         DATA   BUFF     INPUT BUFFER ADDRESS
         DATA   >0A      10 CHARACTERS MAX INPUT
         DATA   0        NUMBER OF CHARACTERS ACTUALLY INPUT
         DATA   TABLE    VALIDATION TABLE ADDRESS
         BYTE   >80      FIELD START POSITION FLAG SET
FLAG2    BYTE   >84      CURSOR POSITION AND VALIDATION
                         FLAGS SET
         BYTE   0        FILL CHARACTER
         BYTE   0        EVENT BYTE
         BYTE   1        CURSOR IN ROW 1
         BYTE   2        COLUMN 2
         BYTE   1        FIELD IN ROW 1

         BYTE   2        COLUMN 2
MASK2    BYTE   >2
```

Perform the read operation as follows:

```
XOP     @RASCI,15
```

If field validation is performed by the task (that is, checking if the first character input is a letter A–Z), the Field Begins in Error flag is set when the input is invalid and the call is reissued.

```
SOCB    @MASK2,
        @FLAG2
XOP     @RASCI,15
```

9.6.7 Hard Break Key Sequence

A hard break key mechanism is necessary to abort errant tasks. To perform the hard break key sequence, you press the Attention key, release it, and hold down the Control key while you press the X key. A foreground task may elect to process the hard break itself by providing end action; otherwise, the task will be aborted and SCI will be unsuspended. Any pending I/O requests at the terminal are aborted, therefore the hard break sequence should be used with caution.

The hard break is only applied against the task initiated by the .BID primitive and its descendants, and never affects SCI itself (refer to the *DX10 Systems Programming Guide* (Volume V) for information on the .BID primitive). In fact, the hard break does nothing if a foreground SCI is not active at the terminal or if SCI is not in task state >17, i.e. suspended waiting for the task that it bid to terminate.

A foreground task which will processs the hard break key sequence must specify end action. The first instruction in the end action routine should be a LWPI to preserve the interrupted workspace. A Get End Action Status SVC is executed to get the interrupted WP, PC, ST and a task error code. If this code has a newly defined value of >10, the task processes the break key interrupt, and puts the WP, PC, and ST in registers 13, 14, and 15, respectively. A RTWP instruction will return to the main program after issuing the Reset End Action SVC. Subsequent break sequences entered before the task end action is entered are ignored. Another hard break key sequence entered during end action and prior to the Reset End Action SVC will abort this task.

9.7 OTHER I/O RELATED CALLS

In addition to the I/O calls that use the SVC code >00 in byte 0, DX10 also supports the I/O related SVCs discussed in the following paragraphs. Each of these I/O related SVCs requires the indicated SVC code in byte 0 of the call block.

9.7.1 >01 — Wait for I/O SVC

The Wait for I/O SVC places the calling task in suspension pending completion of a specified I/O operation. The SVC call block contains four bytes, aligned on a word boundary.

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >01.
1		Initialize to zero. System returns any error.
2–3		Contains the address of the call block which defines the I/O operation, plus 2.

When the specified operation is not in progress, the function returns control to the calling task immediately.

The following example shows program coding for the Wait for I/O SVC

```
         EVEN
SCBS     BYTE 1,0          Suspend the calling task pending
         DATA BLOCK5+2     completion of the I/O operation defined in the call block at
                           location BLOCK5.
```

9.7.2 >30 — Get Event Key by ID SVC

The Get Event Key SVC returns a single event character. The event character is returned in byte 2 of the call block, and the equal bit in the status register is set to one. If there are not event characters in the buffer, the equal bit is set to zero, and no characters is returned.

The Get Event Key SVC accesses a character buffer by ID or by LUNO. To access the buffer by station ID, specify SVC code >30 in byte 0 of the call block, and code the remainder of the call block as described below. To access the buffer by LUNO, specify SVC code >39 in byte 0 of the call block.

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >30 (or >39)
1		Initialize to zero. System returns the status code as follows: 00 = Event character returned 01 = Buffer Empty 02 = Station not available 03 = Illegal station number (or LUNO not assigned)
2		Initialize to zero. Event character is returned in this byte.
3		Supply the station ID (or LUNO) in this byte. If you are using >30 in byte 0, you must specify a station ID here. If you are using >39, you must specify a LUNO here.

9.7.3 >36 — Wait on Multiple Initiate I/O SVC

This call allows a task to suspend until any one of several initiate I/O calls pending for the given program is complete. When the task is unsuspended, it must examine the busy flag in each call block to determine which I/O operation has completed. If no I/O is outstanding for the requesting task, the system returns control immediately.

Byte	Meaning
0	Contains the SVC code. Must be >36.
1	Set to 0
2	Reserved. Set to 0.

Currently there are no error codes for SVC >36.

The following example shows program coding for the >36 SVC call:

```
SCBB BYTE >36,0,0
```

9.7.4 >39 — Get Event Key by LUNO SVC

The Get Event Key by LUNO SVC operates exactly like the Get Event Key by station ID SVC (code >30), and uses the same call block described with that SVC. You must specify >39 in byte 0 and a LUNO number in byte 3 to access the buffer by LUNO. Refer to SVC >30 for more information.

9.7.5 >0F — Abort I/O on Specified LUNO SVC

The Abort I/O supervisor call, code >0F, terminates I/O operations on the specified device. The calling task is suspended during execution of the Abort I/O supervisor call. If the device is file-oriented, it is closed. If the device is busy, the system sets the error flag (I/O call) for the current operation. No device operation is performed, and the medium remains positioned as the last I/O operation left it; i.e., the tape in a cassette is not backspaced or rewound, nor are the remaining cards of a deck read.

If another program has I/O queued for a device, and this SVC is executed for the same device, the current output and all queued I/O output for that device are terminated, and each I/O is returned to the task with an error >06 (I/O error).

The supervisor call block consists of two bytes, and need not be aligned on a word boundary. Byte 0 contains the SVC code (>0F), and byte 1 contains the LUNO assigned to the device.

The following example shows the coding for a supervisor call block for the Abort I/O call:

```
SCBA BYTE >0F,>11          Abort I/O to the device to which LUNO >11 is assigned.
```

NOTE

This supervisor call is available only to privileged tasks, since it can affect other tasks.

9.8 PASS THRU MODE

A special mode of operation on the 931 VDT and 940 EVT is the PASS THRU mode. The PASS THRU mode allows a task to gain direct control of a 931 or 940 terminal. The system provides the necessary communication facilities for the task, with no conversion, mapping, or edit functions provided. A Write ASCII operation transmits the number of characters (up to 80) specified in the physical record block (PRB) directly to the 931/940. The system provides parity. A Read ASCII operation retrieves the specified number of characters (up to 86) from the 931/940. Parity is checked and stripped.

This mode has the following restrictions:

- It does not honor the HOLD function (PREV FORM key) in the normal manner. This implies that the hard break key function is ignored.

- It can cause loss of data received from the 931/940 if a command is sent that causes the 931/940 to transmit a large amount of data. A KSB buffer overflow can occur. The size of the KSB buffer can be increased by regenerating your system.

The PASS THRU mode can cause problems if used in a concurrent access environment. The mode places the DSR into a state where concurrent tasks accessing the terminal must operate in the same manner. This can cause undetermined results or errors if two or more tasks are accessing the terminal at the same time.

A Close or Open SVC places the 931/940 into its normal mode of operation. Writing the edit flags with the PASS THRU mode turned off also places the 931/940 into its normal mode of operations. Any characters in the device KSB buffer and FIFO area are lost at this time.

9.9 EDIT FLAG WORDS

Edit flags are available for changing certain operational characteristics of the DSR. These flags are returned as part of the information returned on a read device status (subopcode >05) operation. The flags can be modified by using a subopcode >15 (Modify Device Characteristics) operation. Byte zero contains the SVC opcode, >00, and byte 1 is initialized to zero. The system returns any error in byte 1. The edit flags are as follows:

Byte	Bit	Meaning
2	0	Enables PASS THRU mode. No translation, echoing, or processing. No extended flags are honored. Read operations are terminated when the buffer is full, or special conditions specified in bits 1 and 2. Write operations write the specified number of characters (up to 80). The DSR provides parity. Only Read and Write ASCII subopcodes are allowed.
	1	In PASS THRU mode, terminates read operations when an ETX is received. The ETX is left in the buffer. ETX is equal to >03.
	2	In PASS THRU mode, terminates read operations when the ESC key followed by a) is received. The ESC) is left in the buffer. ESC is equal to >1B.
	3–15	Reserved. Must be zero.
4–5	0–15	Reserved. Must be zero.

You can use subopcode > 15 to modify the edit flag words. The format of the call block is as follows:

Byte(s)	Description	Value
0	SVC Code	0
1	Error Code	0
2	Subopcode	> 15
3	LUNO Number	0
4	System Flags	0
5	User Flags	0
6, 7	Buffer Pointer	Address of buffer containing flag words 1 and 2
8, 9	Logical Record Length	0
10, 11	Character Count	> 0006

The format of the buffer is as follows:

Byte(s)	Description	Value
0, 1	Reserved	0
2, 3	Edit Flag Word 1	As desired
4, 5	Reserved	0

In the following example, the program changes the edit flag words to indicate PASS THRU mode. It also indicates that reads are to be terminated on receipt of an ETX (> 03):

```
       EVEN
       BYTE   0,0     I/O STATUS
       BYTE   >15     MODIFY DEVICE CHARACTERISTICS
       BYTE   0       LUNO TO BE FILLED IN
       BYTE   0,0     SYSTEM, USER FLAGS
       DATA   BUFFER  ADDRESS OF BUFFER
       DATA   0       LOGICAL RECORD LENGTH
       DATA   6       CHARACTER COUNT
       DATA   0       RESERVED
BUFFER DATA   0       RESERVED
       DATA   >C000   EDIT FLAG 1-PASS THRU, RETURN ON ETX
       DATA   0       RESERVED
```

10

File I/O SVCs

10.1 INTRODUCTION

DX10 provides disk file I/O support for application and system programs. Disk file I/O is performed through the same SVC call mechanism used to perform I/O to devices. Assembly language programs must use SVCs to perform both file and disk I/O. (In high-level language programs, the language translator generates the appropriate SVCs in response to certain language commands; programmers need not use SVCs directly.)

This section describes file I/O SVCs available to the assembly language programmer. The discussion includes the differences between the key-indexed file I/O call block and the sequential and relative record file I/O call block.

DX10 supports three different file organization structures:

- Sequential files
- Relative record files
- Key indexed files

Section 3 of this manual discusses file organization and usage for each of the three file types. Additional material describing system management of KIFs is included in this section along with the SVCs unique to KIFs.

10.2 FILE I/O SVCS

Sequential and relative record files share the same set of file I/O subopcodes. Most of these subopcodes are the same as those used for device I/O discussed in Section 9, and the effects of the operations are somewhat similar. KIFs share most of these subopcodes, and in addition can access several SVCs unique to KIF operations, such as the Read by Key or Read Current SVC.

The SVC call block required for sequential and relative record file I/O is somewhat different from the SVC call block required for KIF I/O. The sequential/relative record file I/O call block and related SVCs are discussed first in this section, with the KIF I/O call block and related SVCs discussed later.

As with all I/O SVCs, the SVC opcode in byte 0 of both call blocks must always be >00.

10.2.1 Sequential and Relative Record File I/O Call Block

The call block for sequential and relative record files always starts on an even byte boundary. From 4 to 24 bytes are required depending on the call. Figure 10-1 illustrates the sequential/relative record file I/O call block and Table 10-1 indicates the meaning of each field in the call block.

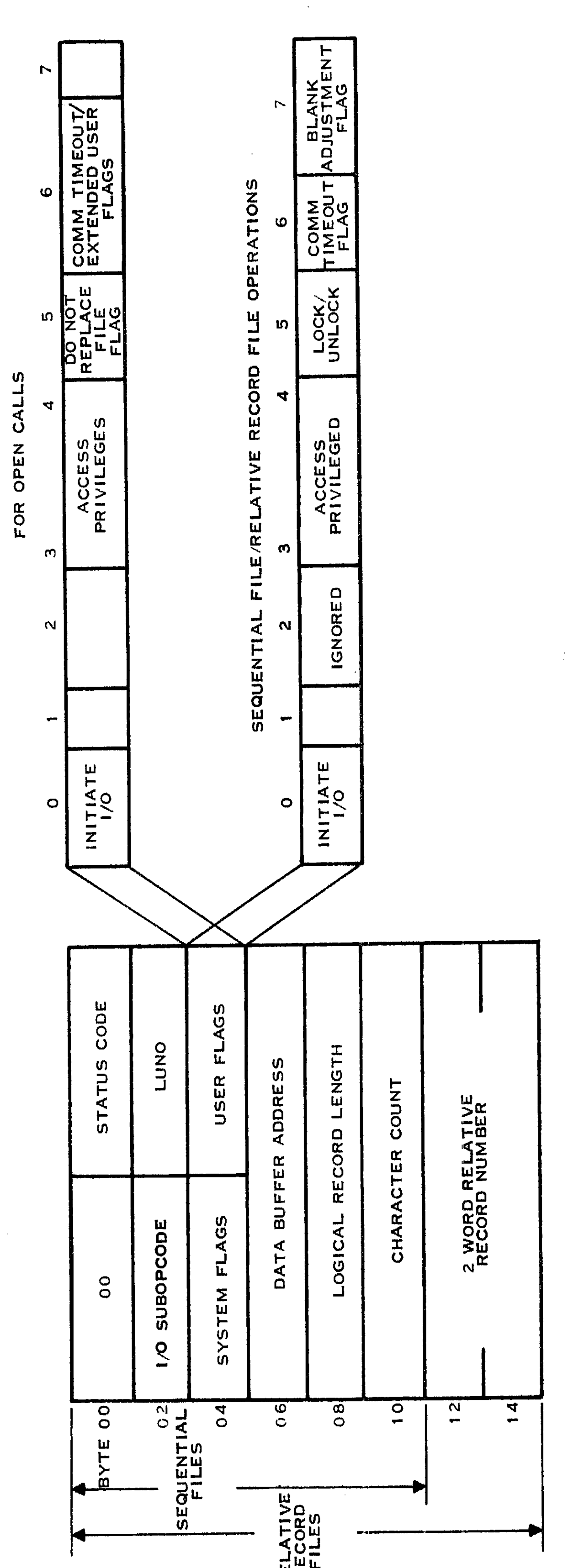


Figure 10-1. Sequential and Relative Record File I/O Call Block

Table 10-1. File I/O SVC Call Block Bit Assignments

Byte	Bit	Meaning
0		The I/O SVC opcode. Must be >00 for all I/O calls.
1		Used by the system to return status code.
2		Contains the subopcode as follows:

I/O Subopcode	Function
>00	Open LUNO
>01	Close LUNO
>02	Close, Write EOF
>03	Open Rewind
>04	Close Unload
>05	Read File Characteristics
>06	Forward Space Record
>07	Backward Space Record
>08	Not Used
>09	Read ASCII
>0A	Read Direct
>0B	Write ASCII
>0C	Write Direct
>0D	Write EOF
>0E	Rewind
>0F	Unload
>10	Rewrite
>11	Modify Access Privileges
>12	Open Extend
>4A	Unlock Record

Byte	Bit	Meaning
3		LUNO (Logical Unit Number). The LUNO must have been previously assigned to the file.
4		System flags (set and reset by DX10).
	0	Busy flag; 1 = busy 0 = done. Do not modify data buffer or call block when busy flag is set to 1.
	1	Error flag; 1 = error, 0 = no error. Error code returned in byte 1 as the status code.
	2	EOF flag; 1 = end-of-file record encountered.
	3–7	Reserved.

Table 10-1. File I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
5		User flags. Use these flags to indicate the following to the system:
	0	Initiate flag. When this flag is set, the system returns control to the calling task without waiting for I/O completion.
	1	Reply flag. Ignored for relative record or sequential files. Applicable for key indexed files.
	2	Reserved.
	3, 4	Access Privileges. On open and modify access operations, use these bits to define access privileges. These access privileges are described in Section 3, and summarized as follows: **Code** — **Meaning** 00 — Exclusive WRITE 01 — Exclusive ALL 10 — Shared (for relative record files and KIFs, all programs can read and write; for sequential files, all programs can read and rewrite but not write) 11 — Read Only
	5	Do Not Replace Flag for open operations, Lock/Unlock Flag for read/write/rewrite operations. On an open operation, this flag set to 1 allows an open with write access privilege on a file only if the file was created during the assign LUNO operation by the autocreate operation. If the file existed prior to the assign, the open will fail. On a read operation, this flag set to 1 locks a record, and unlocks it on a subsequent write or rewrite operation. After locking, only the task and LUNO locking the record can access that record. Locked records are unlocked when the LUNO is closed. A task can unlock with the Unlock Record SVC.
	6	Communications Timeout. Not applicable to file I/O. Set to zero.
	7	Blank Adjustment; used for read and write operations on devices with variable length logical records. When this bit is set on read operations, if the input logical record is shorter than the specified input buffer (bytes 8 and 9), the buffer is filled with blanks (> 20) from the end of the input logical record to the end of the buffer. The value returned as the character count (bytes 10 and 11) is then equal to the buffer length.

Table 10-1. File I/O SVC Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
		When this bit is set on write operations, all trailing blanks in the output buffer are deleted before the output is performed. The output character count specified in bytes 10 and 11 is not modified.
6, 7		Data Buffer Address The starting address of the data buffer for read and write operations must be even. On an open operation a file code of >FF, or a device type code is returned in byte 7. A file type is returned in byte 6 as follows: **Code** — **File Type** 0 — Device 1 — Sequential 2 — Relative Record 3 — KIF 4 — Directory 5 — Program File 6 — Image File
8, 9		Record Length. Set by the calling task on read operations to specify the maximum numbers of characters that can be stored in the data buffer. Not used for write operations. If this word is zero for an open operation, DX10 returns the logical record length defined for the file.
10, 11		Character Count. Set by the calling task on write operations to specify the number of characters to be output. Set by DX10 on a read operation to specify the number of characters stored in the input buffer.
12, 15		Record Number. Set by the calling task for relative record files to access a record positioned relative to the first record (record 0). This field is incremented at the end of a successful read or write operation. The field is ignored for sequential files.

10.2.2 Sequential and Relative Record File Operations

The following paragraphs discuss file operations for sequential and relative record files.

10.2.2.1 >00 Subopcode — Open SVC. The open operation assigns a task to a LUNO, enabling the task to perform I/O on the file assigned to the LUNO. If the open is successful, the calling task has exclusive access to the LUNO whether the LUNO is global, station or local. The access privilege granted to the task is the access privilege requested in bits 3 and 4 of byte 5. An open must be performed before a task can access a file.

The call block is 12 bytes long. The following call block parameters are applicable to the Open SVC:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the subopcode. Must be >00.
3		LUNO to be opened.
5		User flags:
	3, 4	Access privileges
	5	Do not replace
6		File type code returned:

Code	File Type
0	Device
1	Sequential
2	Relative Record
3	KIF
4	Directory
5	Program File
6	Image File

Byte	Bit	Meaning
7		File indicator = >FF, or device type code returned.
8, 9		Returns logical record length defined for the file if field is zero.

10.2.2.2 >01 Subopcode — Close SVC. The close operation releases a LUNO from the calling task. The LUNO remains assigned to the file. A close operation does the following:

- Unlocks any locked records associated with the LUNO
- Writes all modified file blocks retained in memory associated with the LUNO on which write was deferred
- Releases access privileges granted to the LUNO

The call block is 12 bytes long. Call block parameters applicable to a close operation are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the subopcode. Must be >01.
3		Contains the LUNO you want to close.

10.2.2.3 >02 Subopcode — Close EOF SVC. The close EOF operation is the same as a write EOF followed by a close.

10.2.2.4 >03 Subopcode — Open Rewind SVC. The open rewind operation is the same as an open followed by a rewind.

10.2.2.5 >04 Subopcode — Close Unload SVC. The close unload operation is the same as a close.

10.2.2.6 >05 Subopcode — Read File Characteristics SVC. The Read File Characteristics operation provides the user access to the file characteristics information. The user must specify the buffer address at which file characteristics are stored in bytes 6 and 7 of the call block and the length of the buffer in bytes 8 and 9 of the call block. The specified buffer must begin on a word boundary. The number of bytes returned by the Read File Characteristics operation will be indicated by bytes 10 and 11 of the call block.

The number of bytes returned for the different file types are as follows:

Sequential	10 bytes
Relative record	10 bytes
Program	12 bytes

The characteristics returned are described in Table 10-2.

Table 10-2. Information Returned for the Read File Characteristics Operation

Byte	Bits	Contents
0,1		File Attribute Flags
	0,1	File Usage Flags 00 No Special Usage 01 Directory 10 Program File 11 Image
	2,3	Data Format 00 Not Blank Suppressed 01 Blank Suppressed 10 Reserved 11 Reserved
	4	Allocation Type 0 Fixed Size File 1 Expandable File
	5,6	File Type 00 Reserved 01 Sequential 10 Relative Record 11 Key Indexed (see KIF SVCs)
	7	Write Protection Flag 0 Not Write Protected 1 Write Protected
	8	Delete Protection Flag 0 Not Delete Protected 1 Delete Protected
	9	Temporary File Flag 0 Permanent File 1 Temporary File
	10	Blocked File Flag 0 Blocked 1 Unblocked
	11	Reserved
	12	Immediate Write Flag 0 Write Buffers when Necessary 1 Write Buffers Every Change
	13–15	Reserved
2,3		Physical Record Length
4,5		Logical Record Length
6–9		End-of-Media for the File (Number of logical records for sequential, relative record files, and key indexed files)
10,11		For Program Files Only: Sector/Blocks (byte 10) Sectors/ADU for Disk Type (byte 11)

10.2.2.7 >06 Subopcode — Forward Space SVC. The forward space operation spaces forward the specified number of logical records or until an end-of-file (EOF) record is encountered. Place the number of logical records to be spaced over in bytes 10 and 11. If an EOF record is not encountered, zero is returned in bytes 10 and 11 of the call block. If an EOF record is encountered, the file is positioned following the EOF record, the EOF flag (byte 4, bit 2 of the call block) is set, and the number of records remaining to be spaced over is returned in bytes 10 and 11 of the call block.

If the file is a relative record file, the record number in bytes 12 through 15 is incremented by the number of records actually spaced forward.

10.2.2.8 >07 Subopcode — Backward Space SVC. The backward space operation skips backward (moves in reverse direction) over the specified number of logical records, or until an end-of-file (EOF) record is encountered. Place the number of logical records to be skipped in bytes 10 and 11 of the call block.

If an EOF is encountered, the file is positioned so the next read operation encounters the EOF record, the EOF flag (byte 4, bit 2 of the call block) is set, and the number of records remaining to be spaced over is returned in bytes 10 and 11 of the call block.

If the file is a relative record file, the record number in bytes 12 through 15 decreases by the number of records actually spaced backward.

10.2.2.9 >09 Subopcode — Read ASCII SVC. The read ASCII operation reads a record of the specified file and stores the data, packed two characters per word, in the buffer at the address specified in bytes 6 and 7. The maximum number of characters in the buffer is placed in bytes 8 and 9. The actual number of characters stored is placed in bytes 10 and 11. This number specifies the number of characters in the record or the value in bytes 8 and 9, whichever is less.

For a sequential file, if an EOF mark is encountered, file management sets the EOF flag (byte 4, bit 2), and sets the character count in bytes 10 and 11 to zero. If an odd length record is read, an extra character is placed in the buffer but the actual (odd) number of characters in the record is still placed in bytes 10 and 11. When the file is a relative record file and the read operation is successful, file management increments the record number in bytes 12 through 15. If you request a record number past the EOF, the EOF bit gets set, and the record number is not incremented. This condition does not produce an error.

10.2.2.10 >0A Subopcode — Read Direct SVC. The Read Direct SVC works identically to the Read ASCII SVC (subopcode >09).

10.2.2.11 >0B Subopcode — Write ASCII SVC. The write ASCII operation transfers the data in the buffer at the address in bytes 6 and 7 to the specified file. The characters in the buffer are packed two per word. Bytes 10 and 11 contain the number of characters to be written. When the file is a relative record file and the write operation is successful, file management increments the record number in bytes 12 through 15. A Write ASCII operation to a sequential file establishes a new end-of-medium and renders any subsequent records inaccessible.

10.2.2.12 >0C Subopcode — Write Direct SVC. The Write Direct SVC works identically to the Write ASCII SVC (subopcode >0B).

10.2.2.13 >0D Subopcode — Write EOF SVC. The write EOF operation generates an end of file mark in the file. Any number of EOF marks can be written to a sequential file. This allows a file to be divided into subfiles. However, relative record files maintain only one EOF mark. The write EOF operation in a relative record file moves the EOF mark to the record number specified in the physical record block (PRB). If this shortens the file, any records past the EOF are inaccessible and the disk space that they occupy still belongs to the file. (Refer to Section 3 for more information.)

10.2.2.14 >0E Subopcode — Rewind SVC. The rewind operation simulates the rewinding of a magnetic tape file, causing the next operation performed on the file to access the first record in the file (not a subfile), when the specified file is a sequential file. When the file is a relative record file, file management places a zero in the record number field. When the file is an indexed file, the operation is ignored.

10.2.2.15 >0F Subopcode — Unload SVC. The unload operation is ignored for all types of files.

10.2.2.16 >10 Subopcode — Rewrite SVC. The rewrite operation backspaces a sequential or relative record file one record, and writes a record to replace the record previously read. The write portion of the operation is similar to the write ASCII function. If the character count indicates that the updated record is not the same length as the record on file, the operation is terminated, an error code is returned, and the record is not rewritten. Rewrite operations on blank-suppressed files are allowed only when the length of the record remains the same.

The rewrite operation does not alter an end-of-file. When a rewrite operation is attempted on the end-of-file record the rewritten record is lost.

10.2.2.17 >11 Subopcode — Modify Access Privileges SVC. This operation modifies the current access privileges to a new value. When the new access privileges cannot be granted, the old access privileges are retained. The call block is 12 bytes long. Specify the new privileges in bits 3 and 4 of byte 5.

10.2.2.18 >12 Subopcode — Open Extend SVC. The open extend operation opens a file, positioning it at the last EOF in the file. If the file ends with multiple EOFs, the file is positioned to the first EOF of the ending group of EOFs.

If the file is a relative record file, the record number in bytes 12 through 15 is set to the record number of the current end of file.

10.2.2.19 >4A Subopcode — Unlock SVC. The unlock operation releases exclusive control of any previously locked record. The record to be unlocked could have been locked by the unlocking task or any other task. The record to be unlocked is specified in bytes 12 through 15 for relative record files and is the current record for sequential files.

10.2.3 Key Indexed Files

A key indexed file (KIF) is a disk resident structure used for data storage and retrieval in a manner unique to this type of file. Each entry of data made to the file is called a record. The characteristic separating KIFs from other file types is that the records of other files are read by identifying their position in the file, while the records of KIFs are read by identifying a portion of the content of the record. Thus, a user need not search the file for the desired record, only identify it by content.

General KIF usage is discussed in Section 3. The following paragraphs provide additional information helpful when using KIF I/O SVCs. The KIF I/O call block and description of KIF SVCs follows this discussion.

10.2.3.1 Key Indexed File Keys. A portion of a record used to identify the record is called a key. A key is defined at the file level, and applies to every record in the file. A key is a static set of values that cannot change except by reconstructing the file. A key indexed file must have at least one key; otherwise there is no way of identifying the records. In DX10, the maximum number of keys possible is 14. For this reason KIFs are sometimes referred to as *multi-key indexed files* in DX10.

The first key defined becomes the primary key, with the others defined as secondary keys. The primary key need not identify the first portion of a record. Secondary key fields can be located physically before the primary key within a record.

Key indexed file keys are defined at file creation time (see the CFKEY SCI command), and can be from one to 100 characters long. A key has two characteristics. It either can or cannot permit duplicates, and it either can or cannot be modifiable.

If the key value must be unique throughout the entire file, the key must be defined as not permitting duplicates. This restricts any record from being inserted into the file if a record already exists in the file with a key value equal to the key value of the new record. For example, keys such as employee numbers and social security numbers should not be duplicatable, while keys including names and salaries should permit duplicates.

NOTE

> If part of a key has many duplicates (for example, 1,000 or more), inserts, deletes, and rewrites for that key value may require an excessive amount of time. The time on the average is proportional to the number of duplicates of a key value.

You can specify whether or not to allow a key to be modified. A key should be modifiable if its value can change after it is inserted in the file. If the value of an unmodifiable key contains incorrect data when entered into the file, the only way to correct it is by deleting the record and then reentering it. The primary key is always unmodifiable. If a key is modifiable and the first byte contains >FF or the key value is all blanks, the key is a null key. A key containing all binary zeroes is not null.

When the key value for a modifiable key is null (that is, contains all blanks or >FF in the first byte), the record is not catalogued in the index for that key. If a key is modified (on a rewrite) so the first byte contains >FF or blanks, the key becomes a null key.

10.2.3.2 Key Indexed File Records. As records are entered into a key indexed file, they are logically sorted by each key. The insertion process appears as if there is a separate copy of each record for each key of the record. The copies are sorted by each key's value using a given character code, such as ASCII. If more than one record has the same key value, they are sorted in the order they are entered.

The records of a key indexed file are read either randomly or sequentially. When records are read randomly, the file must be opened in random mode, and a key number and key value must be given for each read operation. When the file is read sequentially, a key number and key value is only given the first time. The records can then be read in sorted sequential order either forward or backwards. The records are sorted with respect to the key used in the initial read.

10.2.3.3 Key Indexed File Key and Record Example. Since a key is defined for all records in the key indexed file, the records need to contain similar information, at least for the key portion. Similar information, for example, could be an identifying number or a name. A key indexed file record, and the keys the record can be broken into, are illustrated in the following example.

1-9	10-20	21-30	31-40	41-46	47	48-52
111111111	DOE	JOHN	ANDREW	111111	M	01111

Key	Columns	Definition
1	41-46	Employee Number
2	01-09	Social Security Number
3	10-20	Last Name
4	21-30	First Name
5	31-40	Middle Name
6	10-40	Full Name
7	47-47	Sex
8	48-52	Monthly Salary

The record is 52 characters long, containing seven fields, but because the name fields are used in more than one key, there are eight keys. In the example, the end of one key touches the beginning of the next. This is not a requirement. For data entry purposes, it is easier and clearer to leave one or more blanks between the keys. Although in the example, every column of the record is defined to be in at least one key, it is not required. Quite often only a small portion of the record is defined to be part of a key, while the rest of the record contains data. The only requirements are that there be a primary key, and that no key is longer than 100 characters.

Since the primary key need not be in any particular position in a record or have any qualities different from other keys, there is no way to determine which key is the primary key just by looking at a key indexed file record. However, the primary key can be determined by doing a Map Key Indexed File (MKF) SCI command. The primary key will be identified as key number one.

10.2.3.4 Key Indexed File Algorithm. The system uses a sequential placement scheme to place and retrieve records for a key indexed file. (For earlier releases, hashed placement is still supported.)

The sequential placement scheme is often faster and requires less disk space for a file than other placement schemes. There is no special handling of the primary key, so a Read by Primary Key operation accesses the record at the same speed as the Read by Key operation.

10.2.4 Key Indexed File Call Block and Currency Blocks

The SVC call block for key indexed files shown in Figure 10-2 is almost identical to the call block used by other file types (Figure 10-1). The only items that make the call block for key indexed files different from the call block for sequential or relative record files, is that bit 1 of the user flags field is defined to indicate a key is provided, and bytes 12 and 13 contain a pointer to a block of information known as the currency block.

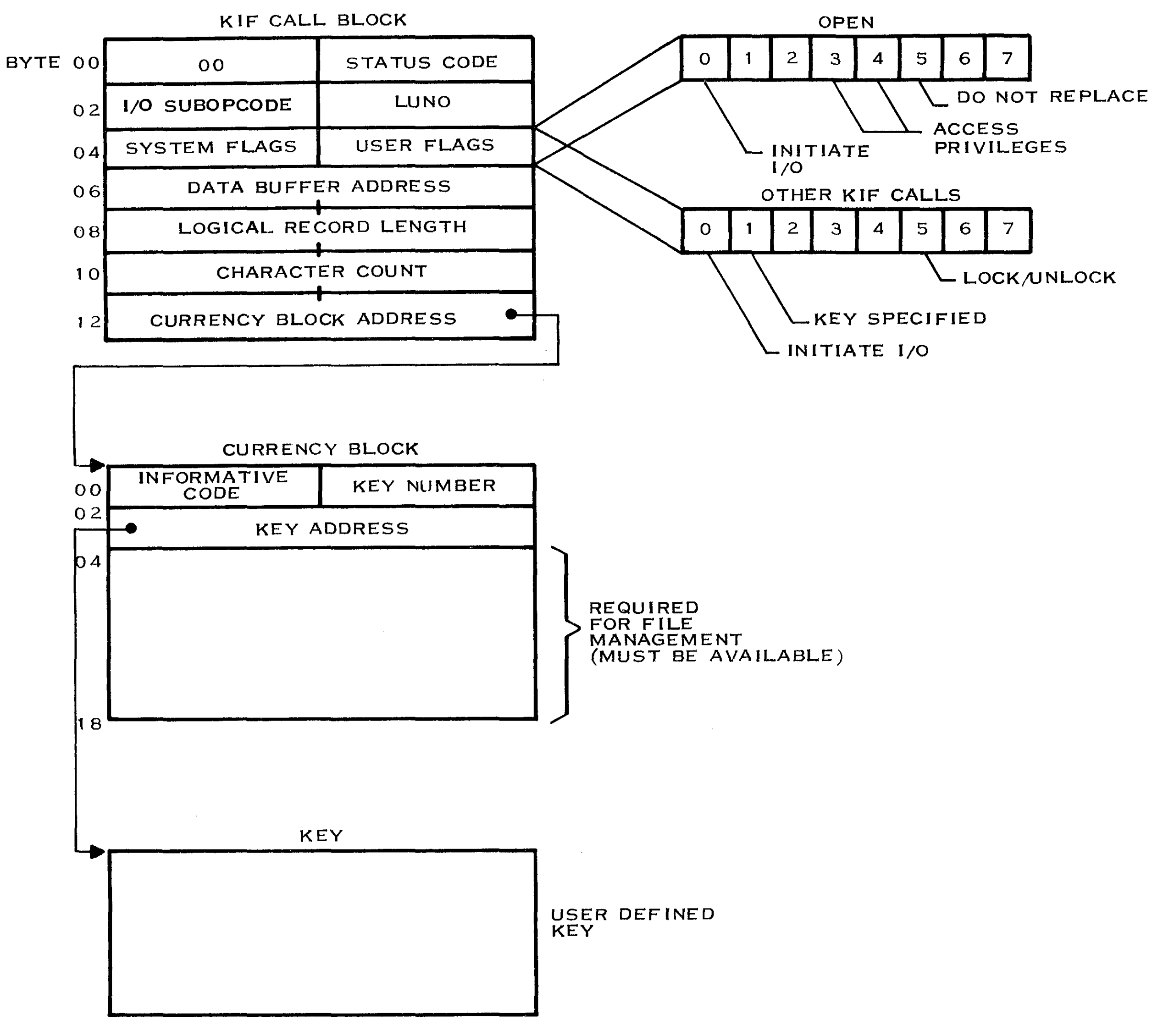


Figure 10-2. KIF Call Block, Currency Block, and KEY Relationship

Currency is the name given to the block of information describing a logical position within a key indexed file. This information includes an informative code, a key number, a pointer to another block, and 16 bytes defined and used by the operating system. The pointer is a set of characters equal in length to the length of the key whose number is indicated in the currency block. The set of characters can either be or not be an actual value of a key in the file. In this manual it is called a key value.

In general, the currency is set so the record read by a read next operation will be the one following whatever record is affected by the previous operation performed. For example, if the same currency blocks are used, a read next operation after an insert operation will get the record following the inserted record. After a delete operation, it will get the record following the deleted record; and after a read operation, it will get the record following the one previously read.

10.2.5 Key Indexed File SVC Subopcodes

Key indexed files support a set of I/O subopcodes unique to this file type as well as some of the subopcodes used by other file types. Except for the close operation, the common subopcodes work only with sequential placement KIF files. The common subopcodes shared by key indexed files and other file types are as follows:

- Open (>00)
- Close (>01)
- Open Rewind (>03)
- Read File Characteristics (>05)
- Forward Space (>06)
- Backward Space (>07)
- Read ASCII (>09)
- Read Direct (>0A)
- Rewind (>0E)

The SVCs and subopcodes unique to KIFs are as follows:

- Open Random (>40)
- Read Greater (>41)
- Read by Key/Read Current (>42)
- Read Greater or Equal (>44)
- Read Next (>45)
- Insert (>46)

- Rewrite (>47)
- Read Previous (>48)
- Delete by Key/Delete Current (>49)
- Unlock Current (>4A)
- Set Currency Equal (>50)
- Set Currency Greater or Equal (>51)
- Set Currency Greater (>52)

In the following paragraphs, the common subopcodes are discussed first, followed by a discussion of the subopcodes unique to key indexed files. In the paragraphs discussing unique subopcodes, the letter *C* preceding the byte number indicates the byte is in the currency block. A KIF must first be opened using the Open Random SVC (>40) before any other SVC unique to KIFs can be accessed.

10.2.5.1 >00 Subopcode — Open SVC. A key indexed file opening with this subopcode cannot have any of the unique key indexed file operations performed to it. The only operations permitted are Close, Forward Space, Backward Space, Rewind, and Read ASCII. The records are accessed in sorted order by the primary key. For instance, when a file opens, a few records are read or skipped over, and then the file is closed and reopened; the logical position in the file is the same as it would have been if the file had not been closed and reopened. That is, the open operation does not automatically position to the logical beginning of a file, unless it is the first open operation for the specified LUNO. (To perform SVCs unique to KIFs, open using the Open Random SVC (subopcode >40).

The fields of the call block used by the Open SVC are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Subopcode. Must be >00.
3		Contains LUNO assigned to the key indexed file.
5	3, 4	Contains access privileges as follows: 00 = Exclusive WRITE 01 = Exclusive ALL 10 = Shared access 11 = Read only
6		File type code returned (3 for KIF)
7		File indicator returned (>FF for KIF)
8, 9		Initialize to zero. Defined logical record length returned.

10.2.5.2 >01 Subopcode — Close SVC. You can use this operation with both the common open operations and the open random operation (>40). (The Open Random SVC is unique to KIFs, and is discussed later). Use the close operation to ensure data integrity. The fields of the call block used by the Close SVC are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the subopcode. Must be >01.
3		Contains the LUNO assigned to the key indexed file.

10.2.5.3 >03 Subopcode — Open Rewind SVC. A key indexed file opened with this subopcode cannot have any of the unique operations performed to it. The only operations permitted are Close, Forward Space, Backspace, Rewind, and Read ASCII. The operation sets the record number to the first logical record in the file; the record with the *smallest* primary key. As in the open operation, records are sorted by the primary key.

The fields of the call block used by the open rewind operation are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero.
2		Contains the SVC subopcode. Must be >03.
3		Contains the LUNO assigned to the key indexed file.
5	3, 4	Contains access privileges as follows: 00 = Exclusive WRITE 01 = Exclusive ALL 10 = Shared access 11 = Read Only
6		File type code returned (3 for KIF)
7		File indicator returned (>FF for KIF)
8,9		Initialize to zero. Defined logical record length returned.

10.2.5.4 >05 Subopcode — Read File Characteristics SVC. The Read File Characteristics operation provides the user access to the file characteristics information. The user must specify the buffer address at which file characteristics are stored in bytes 6 and 7 of the call block and the length of the buffer in bytes 8 and 9 of the call block. The specified buffer must begin on a word boundary. The number of bytes returned by the Read File Characteristics operation is indicated in bytes 10 and 11 of the call block.

For KIFs, the number of bytes returned is as follows:

12 + 4N where N = number of keys (less than 14)

The characteristics returned are described in Table 10-3.

Table 10-3. Information Returned for Read File Characteristics Operation

Byte	Bits	Contents
0, 1		File Attribute Flags
	0, 1	File Usage Flags. Will be 00 for KIF.
	2, 3	Data Format 00 Not Blank Suppressed 01 Blank Suppressed 10 Reserved 11 Reserved
	4	Allocation Type 0 Fixed Size File 1 Expandable File
	5, 6	File Type. Will be 11 for KIFs.
	7	Write Protection Flag 0 Not Write Protected 1 Write Protected
	8	Delete Protection Flag 0 Not Delete Protected 1 Delete Protected
	9	Temporary File Flag 0 Permanent File 1 Temporary File
	10	Blocked File Flag 0 Blocked 1 Unblocked
	11	Reserved
	12	Immediate Write Flag 0 Write Buffers when Necessary 1 Write Buffers Every Change
	13–15	Reserved
2, 3		Physical Record Length
4, 5		Logical Record Length
6–9		End-of-Media for the File (Number of logical records).

Table 10-3. Information Returned for Read File Characteristics Operation (Continued)

Byte	Bit	Contents
10, 11		Number of Keys for the File
12+4(N−1)		Key Flags
	0–4	Reserved
	5	Key Modifiable Flag (Subordinate Keys) 0 Key Not Modifiable 1 Key Modifiable
	6	Reserved
	7	Duplicates Flag 0 Duplicate Key Values Not Allowed 1 Duplicate Key Values Allowed
13+4(N−1)		Key Length
14+4(N−1) to 15+4(N−1)		Key Offset

Note:

N ranges from 1 to 14 and is the key number you require.

10.2.5.5 >06 Subopcode — Forward Space SVC. This operation skips forward over a number of logical records from the present record. If the number of records to be skipped exceeds the number of records remaining in the file, the End-of-File flag is set in the call block (bit 2 of byte 4), and the file is logically positioned before the end-of-file.

The records are sorted by the primary key.

The fields of the call block used by the forward space operation are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >06.
3		Contains the LUNO assigned to the key indexed file.
4	2	System returns a 1 in this bit if EOF was encountered.
10,11		Contains the number of logical records you want to skip over. System returns the number of records remaining to be skipped if bit 2 of byte 4 is set to 1, otherwise system returns zero in these bytes.

10.2.5.6 >07 Subopcode — Backward Space SVC. This operation skips backwards over a specified number of logical records from the present record. The records are sorted off the primary key. If the beginning of the file is encountered, no error is returned and bytes 10 and 11 will contain the number of records remaining to be skipped. A nonzero value in bytes 10 and 11 is the only indication that the beginning of file was encountered.

The fields of the call block used by the Backward Space SVC are:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >07.
3		Contains the LUNO assigned to the key indexed file.
10, 11		Contains the number of logical records you want to skip. If the beginning of the file is encountered, the system returns the number of logical records remaining to be skipped in these bytes. Otherwise, zero is returned.

10.2.5.7 >09 Subopcode — Read ASCII SVC. Operates identically to the Read Direct SVC described next, and uses the same call block.

10.2.5.8 >0A Subopcode — Read Direct SVC. This operation transfers the present logical record that the file is positioned to into a buffer and positions the file to the next logical record. The next logical record is the record with a primary key equal to, or next larger than, the primary key of the record transferred into the buffer. If the file is positioned to the logical end-of-file and a Read operation is executed, the end-of-file bit (byte 4, bit 2) in the call block will be set to one.

The call block for the Read Direct and Read ASCII SVCs is as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >0A (or >09).
3		Contains the LUNO assigned to the key indexed file.
4		System returns a one in this byte if EOF is encountered.
6, 7		Contains the address of the buffer into which you want the record transferred.
8, 9		Contains the maximum number of characters you want transferred into the buffer.
10, 11		System returns the actual number of characters transferred into the buffer.

10.2.5.9 >0E Subopcode — Rewind SVC. This operation resets the present record number to the first logical record, which will be the record with the smallest primary key.

The fields of the call block used by the Rewind operation are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >0E.
3		Contains the LUNO assigned to the key indexed file.

10.2.5.10 >40 Subopcode — Open Random SVC. Before you can use any of the SVCs that are unique to KIF files, you must first open the file using the Open Random SVC>

The fields of the call block used by the Open Random SVC are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the subopcode. Must be >40.
3		Contains the LUNO assigned to the key indexed file.
6		File type code is returned (3 for KIF).
7		File indicator is returned (>FF for KIF).
8, 9		Initialize to zero. System returns the defined logical record length.

10.2.5.11 >41 Subopcode — Read Greater SVC. This operation reads the record with the next larger key value than the specified key. If there is more than one record, the record that was inserted into the file first will be read. The record can be locked when it is read by setting bit 5 of byte 5 of the call block to one; setting this bit to zero indicates that locking is not desired.

You can use this SVC with only a partial key, as described later in this section.

The SVC call block required is the same as that used for the Read Greater or Equal SVC (subopcode >44), and is described with that SVC in the following paragraphs.

10.2.5.12 >42 Subopcode — Read by Key/Read Current SVC. The >42 subopcode is used to specify either the read by key or read current operation. The operation to be performed is designated by setting bit 1 of byte 5 in the call block. When that bit is 1, the Read by Key operation is performed. When that bit is 0, the read current operation is performed.

The read by key operation reads the first record that was inserted into the file with the specified key value. The read current operation reads the record identified by the currency information, instead of the record identified by the key number and key value.

The record can be locked when it is read by setting bit 5 of byte 5 of the call block to 1; setting this bit to 0 indicates that locking is not desired.

The fields of the call block used by the Read by Key and Read Current SVCs are as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
2		Contains the SVC subopcode. Must be >42.
3		Contains the LUNO assigned to the key indexed file.
5	1	Set to 1 for Read by Key operation. Set to 0 for Read Current operation.
6, 7		Contains the address of buffer you want to receive the record.
8, 9		Contains the maximum number of characters you want to be read.
10, 11		System returns number of characters actually read.
12, 13		Contains the currency block address.
For Read by Key:		
C 0		System returns the informative code.
C 2, 3		Contains the address of the block that contains the value of the key to be read.
Key		Block of memory pointed to in bytes 2,3 containing the value of the key to be read.
For Read Current:		
C 0-19		Contains currency information set up by the previous command. System returns the informative code in byte 0.

10.2.5.13 >44 Subopcode — Read Greater or Equal. This operation reads the record with the next larger key value than the specified key; or, if one exists, reads the record with a key value equal to the specified key. If there is more than one record, the record that was inserted into the file first will be read. The record can be locked when it is read by setting bit 5 of byte 5 of the call block to 1; setting this bit to 0 indicates that locking is not desired.

You can use this SVC with only a partial key, as described later in this section.

The fields of the call block used by the Read Greater or Equal SVC are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >44 for the Read Greater or Equal operation (>41 for Read Greater)
3		Contains the LUNO assigned to the key indexed file.
5	1	Set to 1. This indicates a key is specified.
	5	Set to 1 if you want to lock the record.
6, 7		Contains the address of the buffer you want to receive the record.
8, 9		Contains the maximum number of characters you want to be read.
10, 11		System returns the number of records actually read.
12, 13		Contains the address of the currency block.
C 0		Contains the length of the specified key value. System returns the informative code in this byte.
C 1		Contains the number of the key you are using.
C 2, 3		Contains the address of the block in which you place the key value to be used.
Key		The block of memory containing the value of the key you want to use.

10.2.5.14 >45 Subopcode — Read Next SVC. This operation reads the next sequential record in the file. The currency information is used by the operation to determine the correct record, and works the same way as for the Read Previous SVC (>48 subopcode). The call block required is the same also, except that the subopcode must be >45 for a Read Next operation. (See subopcode >48 for call block information.)

10.2.5.15 >46 Subopcode — Insert SVC. This operation places a data record into the key indexed file. The record must have a primary key value. Any secondary keys may be null if they are defined to be modifiable. A key is considered to be null if the first byte contains >FF or if it is entirely blank. (All binary zeroes is not null.) A null key is not catalogued in the index.

An informative code of >B4 is returned if a duplicate of a key is found and the key is defined as not having duplicates.

The fields of the call block used by the Insert operation are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >46.
3		Contains the LUNO assigned to the key indexed file.
6, 7		Contains the address of the buffer that contains the record.
10, 11		Contains number of characters in record.
12, 13		Contains the address of the currency block.
C 0		System returns the informative code in this byte.
C 1		Contains the number of the key you want to use to set up the currency indicators.

10.2.5.16 >47 Subopcode — Rewrite SVC. This operation replaces an already existing record with another one that can be smaller or larger than the original. Only keys defined to be modifiable can be changed with the Rewrite subopcode, and the record must have been read with lock. The currency block set up by the read must be used by the Rewrite SVC.

The fields of the call block used by the Rewrite operation are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >47.
3		Contains the LUNO assigned to the key indexed file.
5	5	Set to 0 to rewrite the record and keep it locked. Set to 1 to rewrite the record and unlock it afterwards.
6, 7		Contains the address of the buffer that contains the record.
10, 11		Contains the number of characters in record.
12, 13		Contains the address of the currency block.
C 0		System returns the informative code in this byte.

10.2.5.17 >48 Subopcode — Read Previous SVC. These operations read either the next sequential record in the file from the present record, or the previous record in the file from the present record; the currency information is used to determine the correct record. If the currency information was set up using one of the Set Currency SVCs, the record presently pointed to by the currency information is read, instead of the next sequential or previous record. The record is locked when read by setting bit 5 of byte 5 of the call block to 1; setting this bit to 0 indicates that locking is not desired.

The fields of the call block used by the Read Previous (and Read Next) operations are as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >48 for Read Previous (>45 for Read Next).
3		Contains the LUNO assigned to the key indexed file.
6, 7		Contains the address of the buffer to receive the record.
8, 9		Contains the maximum number of characters you want to read.
10, 11		System returns the number of characters actually read.
12, 13		Contains the address of the currency block.
C 0–19		Contains currency information set up by the previous command. System returns the informative code in byte 0.

10.2.5.18 >49 Subopcode — Delete by Key/Delete Current SVC. The >49 subopcode specifies either the delete by key or delete current operation. Designate the delete by key operation by setting bit 1 of byte 5 equal to one. Designate the delete current operation by setting bit 1 of byte 5 to zero.

The delete by key operation deletes the record with a key value equal to the one specified. If there is more than one record with an equal key value, the informative code > B4 is returned and the record is not deleted. A delete current command must be used if more than one record has a key value equal to the one specified.

The delete current operation deletes the record pointed to by the currency indicators.

The fields of the call block used by the delete by key operation are as follows, with differences indicated:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >49.
3		Contains the LUNO assigned to the key indexed file.
5	1	Set to 1 for Delete by Key to indicate that a key is specified. Set to 0 for Delete Current.
12,13		Contains the address of the currency block.
C 0		System returns the informative code in this byte.
For Delete by Key:		
C 1		Contains the number of the key you want to use.
C 2,3		Contains the address of the block containing the value of the key to be read.
Key		The block of memory containing the value of the key to be used in finding the desired record.

10.2.5.19 >4A Subopcode — Unlock Current SVC. This operation unlocks the record pointed to by the currency indicators.

The fields of the call block used by the unlock current operation are:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >4A.
3		Contains the LUNO assigned to the key indexed file.
12,13		Contains the address of the currency block.
C 0–19		Contains currency information set up by a previous command. System returns the informative code in this byte.

10.2.5.20 >50 Subopcode — Set Currency Equal SVC. The Set Currency Equal SVC sets the currency indicators to the record containing a key value equal to the one specified. It uses the same call block as the Set Currency Greater or Equal SVC, and the Set Currency Greater SVC. This call block is described with the Set Currency Greater SVC (subopcode >52).

You can use this SVC with only a partial key, as described later in this section.

10.2.5.21 >51 Subopcode — Set Currency Greater or Equal. The Set Currency Greater or Equal operation sets the currency indicators to the record containing a key value equal to the one specified, if one exists. Otherwise, the currency indicators are set to the record containing the next larger key value than the one specified.

You can use this SVC with only a partial key, as described later in this section.

10.2.5.22 >52 Subopcode — Set Currency Greater. The Set Currency Greater SVC (>52) SVC sets the currency indicators to the record containing the next larger key value than the one specified.

You can use this SVC with only a partial key, as described later in this section.

The fields of the call block used by the Set Currency Greater (and the Set Currency Equal and Set Currency Greater or Equal) SVCs are as follows:

Byte	Bit	Meaning
0		Contains the SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >50, >51, or >52, depending on the operation desired.
3		Contains the LUNO assigned to the key indexed file.
12,13		Contains the address of the currency block.
C 0		Contains the length of the key you specify. System returns the informative code in this byte.
C 1		Contains the number of the key you want to use.
C 2,3		Contains the address of the block that contains the value of the key to be read.
Key		The block of memory that contains the key value you want.

10.2.5.23 Using Partial Keys. All three of the set currency operations, the Read Greater, and Read Greater or Equal SVCs have a feature that permits specification of key values with fewer characters than the keys are defined to have. These are known as partial keys.

To use one of these operations, the number of characters in the partial key is placed in byte 0 of the currency block and the partial key characters are placed in the key block. The characters of a partial key will be compared to the beginning characters of the specified key. They cannot be used to find a sequence of characters embedded within a key.

An example of partial key usage is if a key is defined to be a person's last name, such as JOHNSON, a partial key can be used within a program to find all last names beginning with J, JO, JOH, and so on.

10.2.5.24 Error and Informative Codes. Values returned in the status code field, byte 1 of the call block, are serious error codes, and if occurring, it is advisable to terminate the application program. The error codes are discussed in the *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI).

The values returned in the informative code field, byte 0 of the currency block, may or may not be serious errors. They are intended to be an aid to the user and can occur in any properly functioning application program.

The informative codes which may be returned to byte 0 of the currency block are:

Value	Meaning
B3	There are no more records to be read.
B4	There are records with the same key value as that used to read the present record.
B5	No record exists with the key or currency values specified.
B7	The record to be locked is already locked.
B8	No record exists that satisfies the conditions set by the subopcode and currency block.
BD	The next record cannot be found for a Read Next or Read Previous operation. Either currency has been destroyed or the record has been deleted.

Table 10-4 identifies the possible error and informative codes for the different key indexed file unique subopcodes. A "*" in front of the list of error or informative codes identifies the codes possible if the operation is performed with the record lock bit on (byte 5, bit 5 of the call block).

Table 10-4. Error and Informative Codes for Key Indexed File Subopcodes

Subopcode	Command	Error Codes	Informative Codes
40	Open Random	—	—
41	Read Greater	05,0D,BF,D7	B4,B8 *B4,B8,B7
42	Read by Key	05,0D,BF,D7	B4,B5 *B4,B8,B7
42	Read Current	05,0D,B6,D7	B4,B5 *B5,B7
44	Read Greater or Equal	05,0D,BF,D7	B4,B8 *B4,B7,B8
45	Read Next	05,0D,BF,D7	B3,BD *B3,B7,BD
46	Insert	05,0D,B1,B2 C0,D7,E0	B4
47	Rewrite	05,0D,B0,B2,B6 B9,BA,BE,C0,E0 *05,0D,B0,B1,B2, *B6,B9,BA,BE,C0, *D7,E0	B5
48	Read Previous	05,0D,B6,D7	B3,BD *B3,B7,BD
49	Delete by Key	05,0D,B0	B4,B5,B7
49	Delete Current	05,0D,B0	B4,B7
4A	Unlock	0D,B6,BA	B5,B7
50	Set Currency Equal	0D,B6,BA	B4,B8
51	Set Currency Equal or Greater	05,0D,D7	B4,B8
52	Set Currency Greater	05,0D,D7	B4,B8

10.2.5.25 Estimating Key Indexed File Size. When key indexed files are created using the sequential placement scheme, the maximum size of the file can be approximately estimated. The accuracy of the final value depends on the accuracy of the values supplied by the user creating the file. The factors affecting accuracy are as follows:

- Average blank suppressed record size.
- Blocking factor for data blocks. The percentage inaccuracy introduced is equal to the reciprocal of the blocking factor times 100. This inaccuracy exists because data records do not occupy more than one physical record in a key indexed file.

  You can add the percentages to get an estimate of the total inaccuracy.

The Create File SVC only bases storage calculation on log blocks and data blocks. It does not account for B-tree blocks. This causes a file created for a given number of records to be larger when actually loaded with that number of records. Refer to the paragraph on the Create File SVC for more information on how to create a key indexed file.

The parameters required for calculating disk usage for a key indexed file are:

- The physical record size
- The average blank suppressed logical record size
- The sizes of all the keys
- The ADU size of the disk on which the file will be created
- The number of logical records (estimate)
- Whether the input will be sorted on one of the keys when the file is initially loaded

The most difficult parameter to estimate is the average blank suppressed logical record size. This is the average size of a logical record if all blanks are removed from all records. The accuracy of this parameter strongly affects the result. The number of logical records can be easily determined if the records exist in a sequential file. Otherwise, this value must also be estimated. The other values are well defined and require no estimations.

The disk allocation of a key indexed file can be broken down into three definite areas. The first is the prelogging area. It is the only area out of the three that has an absolute value. The number of physical records of disk space required for this area is calculated using the following formula:

$$(18 \times K) + 3 = NPRprelog$$

where:

K = number of keys

NOTE

In the following equation, $\lfloor \; \rfloor$ means to round the enclosed calculation down to the nearest integer and $\lceil \; \rceil$ means to round the number up to the nearest integer.

The second area is used for the B-tree nodes. These are the records containing the structures making key indexed files function differently from other file types. The approximate number of physical records required for these structures is defined with the following equations:

$$\left\lfloor \frac{PRS - 20}{KS + 6} \right\rfloor = X$$

$$\left\lceil \frac{\#LR}{X} \right\rceil + \left\lceil SPLIT \times \left\lceil \frac{\#LR}{X} \right\rceil \right\rceil = NPRb\text{-}tree$$

where:

PRS = Physical record size
#LR = Maximum number of logical records
KS = Size of the key
SPLIT = 0.1 if the input is already sorted with respect to the key, otherwise, it equals 0.25

A calculation must be performed for each key to be in the file. Thus, if there are to be three keys in the file, there must be three B-tree values in the final physical record summation.

The last area is used for the actual data records. The approximate number of physical records required for this area is defined by the following equation:

$$\left\lfloor \frac{PRS - 16}{LRS + 6} \right\rfloor = X$$

$$\left\lceil \frac{\#LR}{X} \right\rceil = NPRdata$$

where:

PRS = Physical record size
#LR = Maximum number of logical records
LRS = Average blank suppressed logical record size (if there is only one key in the file, this value should not include the length of the key, i.e., assume the key consists of all blanks)

The 16 bytes subtracted from the PRS is 14 bytes of overhead at the beginning of each record, and 2 bytes at the end of each record.

The total number of physical records required is calculated by the following equation:

$$\text{NPRprelog} + \text{NPRdata} + \sum_{i=1}^{K} \text{NPRb-tree}_i$$

where:

K = Number of keys

The total number of ADUs required is therefore defined through the following equation:

If PRS is greater than, or equal to ADU,

then $$\left\lceil \frac{\text{PRS}}{\text{ADU}} \right\rceil \times \text{NPRtotal} = \text{Number of ADUs required}$$

else APRS = PRS rounded up to the next multiple of the sector size

$$\left\lceil \frac{1}{\left\lfloor \frac{\text{ADU}}{\text{APRS}} \right\rfloor} \times \text{NPRtotal} \right\rceil = \text{Number of ADUs required}$$

where:

PRS = Physical record size
ADU = ADU size

The following are examples of these calculations.

Example 1:

PRS = 864
ADU = 864
LRS = 60 (Average blank suppressed size; key size not included)
KS = 20
K = 1
#LR = 800

Sorted input (SPLIT = .1)

A) NPRprelog = (18 × 1) + 3 = 21

B)
$$\left\lfloor \frac{864 - 20}{20 + 6} \right\rfloor = 32$$

$$\left\lceil \frac{800}{32} \right\rceil + \left\lceil .1 \times \left\lceil \frac{800}{32} \right\rceil \right\rceil = 25 + 3$$

NPRb-tree = 28

C)
$$\left\lfloor \frac{864 - 16}{60 + 6} \right\rfloor = 12$$

$$\text{NPRdata} = \left\lceil \frac{800}{12} \right\rceil = 67$$

NPRtotal = NPRprelog + NPRb-tree + NPRdata
= 21 + 28 + 67
= 116

$$\text{ADUs} = \left\lceil \frac{864}{864} \right\rceil \times 116 = 116$$

Example 2:

PRS = 864
ADU = 864
LRS = 60
KS1 = 20
KS2 = 20
KS3 = 20
K = 3
#LR = 2600
Random input (SPLIT = .25)

A) NPRprelog = (18 × 3) + 3 = 57

B)
$$\left\lfloor \frac{864 - 20}{20 + 6} \right\rfloor = 32$$

$$\left\lceil \frac{2600}{32} \right\rceil + \left\lceil .25 \times \left\lceil \frac{2600}{32} \right\rceil \right\rceil = 82 + 21$$

NPRb-tree(1) = 103 key 1
NPRb-tree(2) = 103 key 2
NPRb-tree(3) = 103 key 3

C)
$$\left\lfloor \frac{864 - 16}{60 + 6} \right\rfloor = 12$$

$$\text{NPRdata} = \left\lceil \frac{2600}{12} \right\rceil = 217$$

NPRtotal = NPRprelog + NPRb-tree(1) + NPRb-tree(2) + NPRb-tree(3) + NPRdata
= 57 + 103 + 103 + 103 + 217
= 583

$$\text{ADUs} = \left\lceil \frac{864}{864} \right\rceil \times 583 = 583$$

10.3 FILE UTILITY SVCS

File utility SVCs perform directory and file maintenance and access operations such as creating files, assigning LUNOs to pathnames, adding aliases, and assigning file protection (write protection, delete protection). They do not involve data transfer to or from records in the files. These utility operations allow a task to acquire its own resources, thereby reducing the operator's workload.

The file utility SVCs and their subopcodes are as follows:

Subopcode	Utility Operation
90	Create file
91	Assign LUNO to pathname
92	Delete file
93	Release LUNO assignment
94	Reserved
95	Assign new filename (rename)
96	Unprotect file
97	Write protect file
98	Delete protect file
99	Verify pathname
9A	Add alias
9B	Delete alias
9C	Define forced write mode

File utility SVCs require a slightly different call block than other SVCs. The utility call block is shown in Figure 10-3 and the byte and bit assignments are given in Table 10-5. Refer to Appendix E for full I/O SVC call block assignments.

The general file utility call block applies to all file types. One of the entries in the utility call block is a pointer to the expansion block required for creating KIFs. (Refer to the Create File discussion later in this section.)

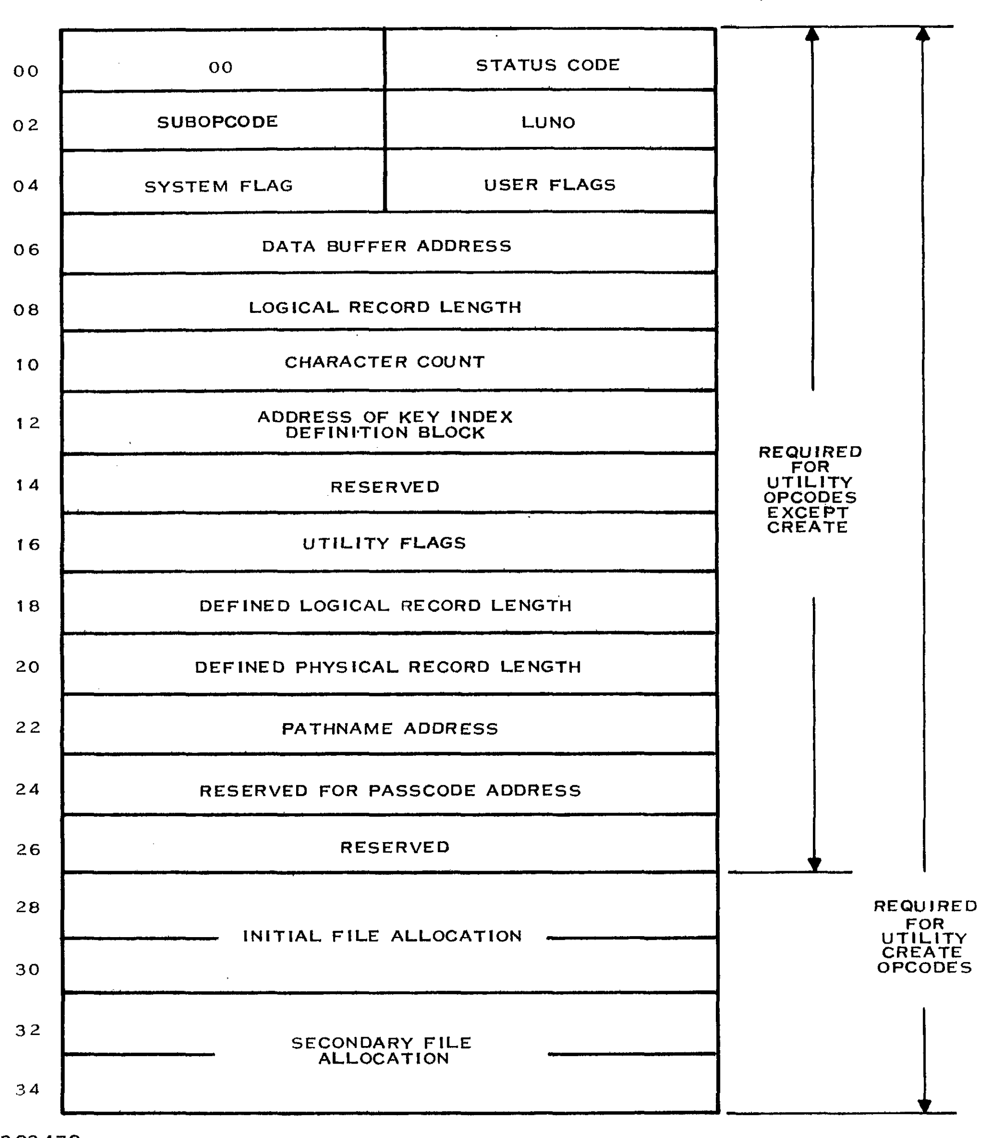


Figure 10-3. File Utility SVC Call Block

Table 10-5. File Utility Call Block Bit Assignments

Byte	Bit	Meaning
0	All	SVC opcode. This code is zero for all I/O operations
1	All	Status code (see the *DX10 Operating System Error Reporting and Recovery Manual* (Volume VI)).
2	All	Utility subopcode. Set to a value between >90 and >9C to specify the desired utility operation.
3		LUNO. This byte specifies the LUNO used in the operation. It is user supplied except when the Generate LUNO bit is set on an Assign operation.
4–11		Not used. Reserved for compatibility with other calls.
12,13		When creating a key indexed file, the information about the number of keys and key positions is defined by an expansion block. This block varies in length depending upon the number of keys in the file. The format of this block is shown in Figure 10-3. For program files, byte 13 contains the maximum number of tasks. When creating a directory file, this value is used to establish the default physical record size that is to be used for subsequent file creations under this directory.
14,15		For program files, the maximum number of procedures is specified in byte 14. The maximum number of overlays is specified in byte 15. These bytes are not used for other types of files.
16		Utility Flags. These flags, and the flags in byte 17, are used for Assign LUNO and Create File operations. For all other utility operations, these flags should be set to 0.
16	0	Created by Assign. This bit is set after an Assign LUNO operation in the Autocreate option (bit 6) was specified and assignment actually caused the file to be created.
	1,2	File Usage flag. The user sets this field to indicate the special usage of this file. For Create operations this indicates that the file will be used for a special purpose. **Code** — **Meaning** 00 — No special usage 01 — Directory file 10 — Program file 11 — Image file

Table 10-5. File Utility Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
	3,4	Scope of LUNO assignment. These bits define the scope of a LUNO assignment or release. **Code** — **Meaning** 00 — The LUNO is task local 01 — The LUNO is station local 10 — The LUNO is global 11 — Reserved
	5	Generate LUNO. If this flag is one, DX10 automatically generates a LUNO number, returning it in the LUNO field (byte 3). If this flag is a 0, the content of the LUNO field is used. This flag applies to assign LUNO operation only.
	6	Automatic Create. User sets to one to request automatic file creation by the assign LUNO operation (>91). If set, the call block must contain sufficient information to permit file creation.
	7	Reserved. Must be zero.
17		Utility Flags
	0	LRL flag. User sets this bit to a one for any Create operation, including Autocreate, when the file is to be created with the logical record length in bytes 18 and 19 of the utility block. If this flag is zero, bytes 8 and 9 are used. This flag is for compatibility only. Users should set this flag to a one, and use bytes 18 and 19 for logical record length.
	1	Temporary File. If this bit is one, the file specified will be a temporary file. This applies to Create and to Assign LUNO operations with the Autocreate option specified. This file is allocated on the system volume if the pathname address is zero. Only sequential and relative record files may be temporary.
	2	Immediate Write. Set to one if all disk file writes are to be written immediately, for example, before returning I/O completion to the user. This bit has meaning for the Create File and Define Force Write Mode operations. Selecting immediate write assures data is transferred to disk on each write but is inefficient on blocked files.
	3,4	Data Format. These bits specify the data format of the records. Valid codes are: **Code** — **Meaning** 00 — Normal record image 01 — Blank suppressed 10,11 — Reserved for new data formats. Key indexed files are always blank suppressed.
	5	Allocation Flag. A one indicates the file can grow beyond the initial allocation requested at file creation.

Table 10-5. File Utility Call Block Bit Assignments (Continued)

Byte	Bit	Meaning
	6,7	File Type. Set to one of the following codes to specify the file type when the file is created. This field is ignored except for create operations. The file type is returned on all open operations. **Code** — **Meaning** 00 — Reserved 01 — Sequential file 10 — Relative record file 11 — Key indexed file
18,19		Logical Record Length. This field is required for all files using fixed-length logical records. Its only use for files with variable length logical records is in calculation of file size when the file is created. In this case if the field is 0, a default value dependent on the file type is used. This applies to create operations only. The logical record size cannot exceed 32K bytes.
20,21		Physical Record Length. This field specifies the physical record length to be used for file I/O. If this field is any value less than twice the logical record length, the file will be unblocked. In variable logical record length files this field specifies the size of the blocking buffer and the physical record to be used. If this field is 0, a default is used. The default value varies depending on the disk type in use. This applies to create operations only.
22,23		Pathname Pointer. These two bytes contain the address of a pathname used for the following operations: rename file, add alias, delete alias, protection operations, assign LUNO, create file, and delete file. The pathname format is as follows: Byte 0 — Pathname length in bytes Bytes 1 through n — Pathname The pathname syntax is described in the *DX10 Operations Guide* (Volume II).
24,25		Passcode Pointer. (To be implemented in a future release.)
26,27		Reserved for Passcode Pointer.
28,29 30,31		Initial File Allocation. For a bounded file this specifies the number of records in the file and for an unbounded file it specifies the initial allocation expressed as a number of records (which may later be expanded). On an unbounded file this field can be zero to specify a default initial allocation. This applies to create operations only.
32,33 34,35		Secondary File Allocation. This field is treated exactly as the Initial File Allocation field except that it is used to specify the number of records to be allocated when an unbounded file is expanded. If zero, a default is used. This applies to create operations.

10.3.1 >90 Subopcode — Create File SVC

Use the >90 subopcode in a task to create a file. This SVC creates all file types supported on DX10. (Section 3 describes file types; an earlier paragraph in this section presents information on calculating space usage on KIFs.)

The Create File SVC uses the general file utility SVC call block, but not all the bytes are required. The following fields of the block need to be specified:

Byte	Bit	Meaning
0		SVC opcode. Must be >00.
1		Initialize to zero. System returns any error code.
2		Contains the SVC subopcode. Must be >90.
3–11		Not used. Set to 0.
12,13		Address of KIF extension call block.
14,15		Not used. Set to 0.
16		Utility flags:
	1, 2	00 = ordinary relative record, sequential or KIF. 01 = directory relative record file. 10 = program relative record file. 11 = image relative record file.
17		Utility flags:
	0	Logical Record Length flag. 1 = LRL specified in bytes (18, 19). 0 = LRL specified in bytes (8, 9).
	1	Temporary flag. 1 = temporary file. 0 = permanent file.
	2	Force immediate write. 1 = immediate write (always set for KIF) 0 = deferred write
	3, 4	Data format. 00 = normal record image. 01 = blank suppressed.
	5	Expandable flag. 1 = expandable. 0 = not expandable.
	6, 7	File type. 00 = device. 01 = sequential. 10 = relative record. 11 = KIF (Note: KIFs require an additional call block described later.)

Byte	Bit	Meaning
18, 19		Logical record length (a 0 specified here causes the system to take the default record length for the associated directory, if specified, otherwise it assumes the default specified during system generation.
20, 21		Physical record length (a 0 specifies default).
22, 23		Pathname address. This field specifies the pathname of the file to be created. For temporary files with system generated filenames specify 0.
28–31		Initial file allocation size. This field specifies the number of records allocated to the file being created. Specify greater than 0 for nonexpandable files (a 0 indicates default).
32–35		Secondary file allocation size. Used only with expandable files to specify number of records for secondary allocations (strictly speaking, specifies first secondary allocation. Successive allocations will be larger.) Zero indicates system default allocation size.

When creating KIFs, you must additionally specify an expansion call block to define the keys required for the new file. This expansion block is shown in Figure 10-1. (The expansion block is the KIF definition call block.) All bytes in the KIF definition call block figure are self-explanatory except the flag bytes. There is one flag byte for each key defined for the file. The bit assignments of the flag bytes are shown in Table 10-6.

KIFs are always expandable. The system computes the initial allocation by using the greater of initial allocation and maximum allocation. It uses the logical record length assuming no blank suppression to compute the amount of disk space to allocate to the file. It does not include any B-tree space, only space for the number of data records specified and log records.

Table 10-6. Key Indexed File Definition Bit Assignments

Byte	Bit	Meaning
8,12, etc.	0–4	Not used. Set to 0.
	5	Set to 1 if this key can be modified. Set to 0 if the key must not be modified and must be present on an insert operation. Primary keys are never modifiable, so this bit is always 0 in byte 8.
	6	Reserved. Set to zero.
	7	Set to 1 to allow duplicates on this key. Set to zero if no duplicates allowed.
		The offset bytes must contain the character position within the record at which the key being defined begins.

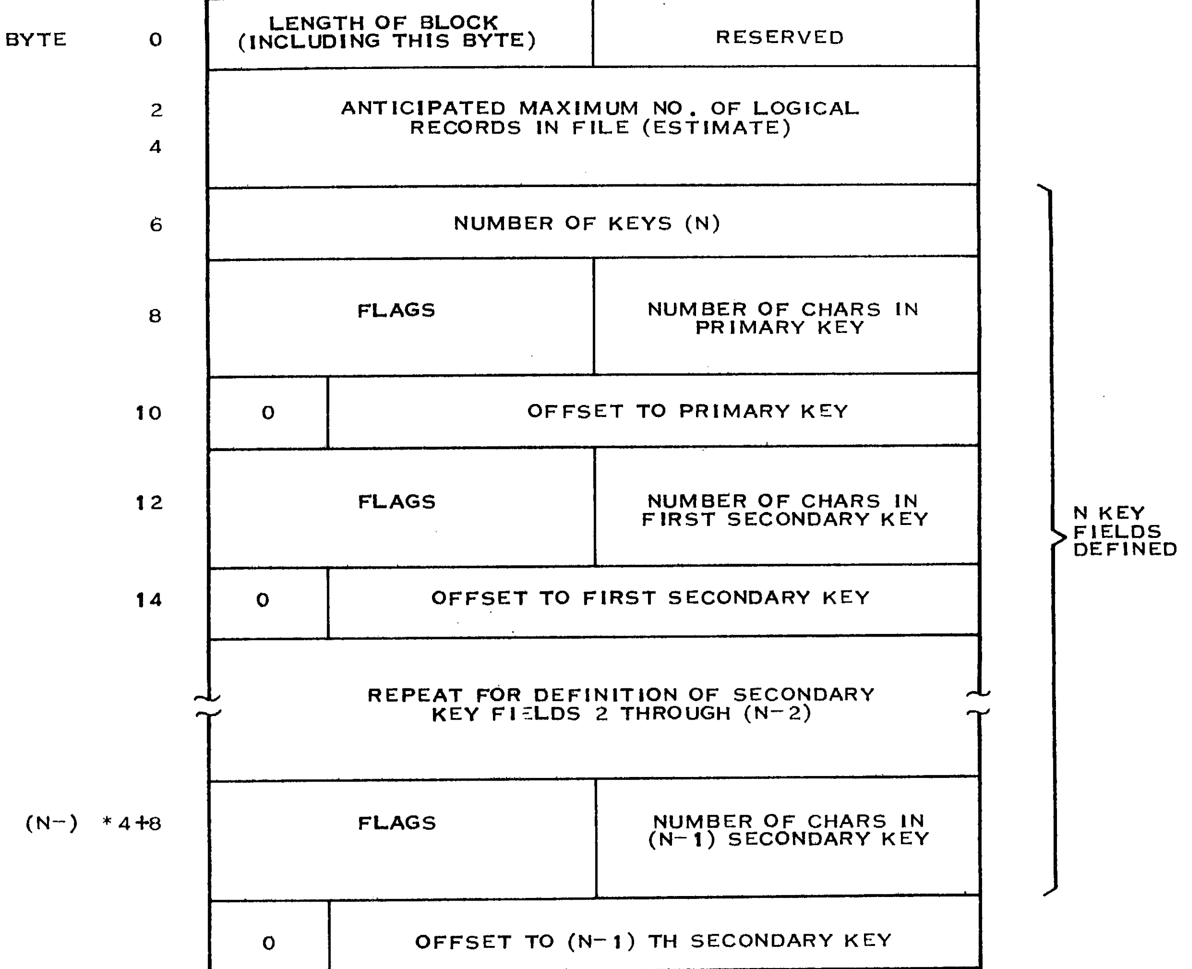


Figure 10-4. Key Indexed File Definition Call Block

10.3.1.1 Create Sequential File Example. The following example creates a call block with these specifications:

- 80-byte logical records
- 288-byte physical records
- 1000-record initial allocation
- 500-record secondary allocation
- blank suppression
- pathname of PACK.USER.TEXT.FILE01

```
*
*EXAMPLE
*
CRSEQ   DATA  0
        BYTE  >90,0,0,0                    CREATE FILE
        DATA  0,0,0,0,0
        BYTE  0,>8D                        LRL,BLANK SUPPRESSED,
                                           EXPANDABLE,
*                                          FILE TYPE.
        DATA  40                           RECORD LENGTH LESS
*                                          AVERAGE NO. BLANKS
        DATA  288                          PHYSICAL RECORD LENGTH
        DATA  PATH                         PATHNAME ADDRESS
        DATA  0,0
        DATA  0,1000                       INITIAL ALLOCATION
        DATA  0,500                        SECONDARY ALLOCATION
PATH    BYTE  NAME-$-1                     PATHNAME LENGTH
        TEXT  'PACK.USER.TEXT.FILE01'      PATHNAME
NAME    EQU $
```

Note that the logical record length has been specified as 40 on the assumption the average record contains 40 trailing blanks.

10.3.1.2 Create Key Indexed File Example. The following program uses the Autocreate operation to create a key indexed file whose attributes are as follows:

- 100-character nominal logical records (average number nonblank characters/record)
- 900-character physical records
- 2 keys — both nonmodifiable and no duplicates allowed
- 10,000 anticipated logical records

```
*
*EXAMPLE
*
TASK    DATA   WS,PC,0
WS      BSS    32
*KEY    DEFINITION BLOCK
KEY     BYTE   KEY$-$,0
        DATA   0,10000          10,000 MAXIMUM LOGICAL RECORDS
        DATA   2                2 KEYS
        DATA   12               1ST KEY 12 CHARACTERS
        DATA   1                1ST KEY BEGINS ON 1ST CHARACTER
        DATA   6                2ND KEY 6 CHARACTERS
        DATA   6                2ND KEY BEGINS ON 6TH CHARACTER
KEY$    EQU    $
PATH    BYTE   PATH$-$-1
        TEXT   '.JB.KEY'
PATH$   EQU    $
CRKEY   DATA   0
        BYTE   >91,0,0,0        AUTO CREATE
        DATA   0,0,0
        DATA   KEY,0            KEY BLOCK ADDRESS
        BYTE   >06              GENERATE LUNO, AUTO CREATE
        BYTE   >8F              LRL, EXPANDABLE KIF
        DATA   100              LOGICAL RECORD LENGTH
        DATA   900              PHYSICAL RECORD LENGTH
        DATA   PATH             PATHNAME ADDRESS
        DATA   0,0
        DATA   0,0                INITIAL ALLOCATION
        DATA   0,0                SECONDARY ALLOCATION
EOT     BYTE   4
        EVEN
PC      XOP @CRKEY,15
        XOP @EOT,15
        END
```

10.3.2 >91 Subopcode — Assign LUNO SVC

Use this SVC to associate a LUNO with a file. If the file does not already exist, the system creates it. (This is called autocreate.) You can either specify a LUNO in the call block, or the system can generate it. There are four combinations of options possible with this operation:

- Preexisting file and specified LUNO.
- Preexisting file and system generated LUNO.
- System generated file and specified LUNO.
- System generated file and system generated LUNO.

If you set the autocreate flag (byte 16, bit 6) to one, and the file does not already exist, then the system attempts to create the file using the fields indicated in the description of the create file (> 90 subopcode). If the creation succeeded without error, the autocreate operation then assigns a LUNO to the newly-created file (either a system-generated LUNO, or a user-specified LUNO).

If the autocreate flag is 0, the operation assigns the LUNO to the existing file; if no file exists, an error results.

Once the file exists (either preexisted or created in earlier steps of operation) the actual LUNO assignment is the same. These fields are relevant to the actual LUNO assignment portion of the operation.

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to 0. System returns any error.
2		Contains the SVC subopcode. Must be >91.
3		Contains the LUNO number.
16		Utility flags:
	0	Created by Assign 0 = file not created 1 = file created
	1,2	File usage 00 = no special usage 01 = directory file 10 = program file 11 = image file
	3,4	Scope of LUNO 00 = task local 01 = station local 10 = global

Byte	Bit	Meaning
	5	Generate LUNO 1 = system generates LUNO 0 = indicates LUNO is specified in byte 3
	6	Autocreate 1 = system creates file if it does not already exist 0 = file must already exist
22,23		Pathname address

Unless the correct file usage flags are set to assign a LUNO to a directory, program, or image file, an error is returned. However, setting these bits does not inhibit assigning a LUNO to a sequential, relative record, or key indexed file.

10.3.3 >92 Subopcode — Delete File SVC

The operation deletes the file whose pathname is addressed by the pathname in bytes 22 and 23. The only restriction is that no LUNOs can be assigned to the file at the time the delete operation is attempted.

10.3.4 >93 Subopcode — Release LUNO SVC

Disassociates a LUNO from a file. The following bytes must be specified in the file utility call block:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to 0. System returns any error.
2		Contains the SVC subopcode. Must be >93.
3		Contains LUNO number.
16	3,4	Scope of LUNO: 00 = task local 01 = station local 10 = global 11 = reserved

This call only releases a LUNO on the level indicated within byte 16. For example, if the call is *release a terminal LUNO 88* and there is none, but there is global LUNO 88 and a task local LUNO 88, neither will be released.

10.3.5 >95 Subopcode — Rename File SVC

This call assigns a new name to a file. You must specify the following in the call block:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >95.
3		LUNO of file to be renamed
5	5	Set to 1 for Do Not Replace option.
22,23		Pathname pointer; the indicated pathname becomes the new name of the file.

If a file already exists with the new name, the Rename operation deletes the previous file before assigning the new name to the specified file. The user may set the 'Do Not Replace' user flag (byte 5, bit 5 of the call block) to prevent deleting the previous file and abort the Rename operation. No other LUNOs may be assigned to either file. Any aliases associated with the renamed file are retained.

10.3.6 >96 Subopcode — Unprotect File SVC

This subopcode disables write and delete protection for a file. Specify the SVC opcode, the subopcode, and the pathname pointer in the call block. (Initialize byte 1 to 0.)

10.3.7 >97 Subopcode — Write Protect File SVC

This subopcode enables write protection for a file. Specify the SVC opcode, the subopcode, and the pathname pointer in the call block. (Initialize byte 1 to 0.)

10.3.8 >98 Subopcode — Delete Protect File SVC

This subopcode enables delete protection for a file. Specify the SVC opcode, the subopcode, and the pathname pointer in the call block. (Initialize byte 1 to 0.)

10.3.9 >99 Subopcode — Verify Pathname SVC

This subopcode does a syntax check of a pathname by attempting to assign a LUNO to the file. You only need to specify bytes 22, 23 (pathname pointer). If the pathname is that of an existing file, the file usage, file type, data format, and allocation type are returned in bytes 16 and 17.

10.3.10 >9A Subopcode — Add Alias SVC

This subopcode assigns an alias (i.e., an alternate name) to a file. (Aliases are explained in Volume II under the Add Alias (AA) SCI command.)

Specify the following in the call block:

Byte	Meaning
0	Contains SVC opcode. Must be >00.
1	Initialize to zero. System returns any error.
2	Contains the SVC subopcode. Must be >9A.
3	LUNO assigned to the file.
22,23	Pointer to pathname (the alias).

10.3.11 >9B Subopcode — Delete Alias SVC

This subopcode deletes an alias.

NOTE

An alias can differ from its associated pathname only in rightmost component, as explained in Volume II.

10.3.12 >9C Subopcode — Define Write Mode SVC

This call defines (or changes) the write mode of a file. User needs to specify:

Byte	Bit	Meaning
0		Contains SVC opcode. Must be >00.
1		Initialize to zero. System returns any error.
2		Contains the SVC subopcode. Must be >9C.
3		LUNO assigned to file.
17	2	Write Mode 0 = deferred write 1 = immediate write

This call changes the write mode only for the specified LUNO assignment; file characteristics do not change.

10.4 TEMPORARY FILES

The use of temporary files and autocreate allows a program to process an input file, generate a temporary file, and at the end of the program replace the input file with the temporary file. Necessary steps to accomplish this are:

- Create the temporary file using Assign LUNO (>91) and specify:
  - GENERATE LUNO
  - AUTOMATIC CREATE
  - TEMPORARY FILE
  - Pathname address is equal to 0 (system disk) or set to the address of a pathname of which the volume name identifies the volume on which the temporary file is to be created.
- Write data to the temporary file using the LUNO returned by DX10. Close the temporary file.
- Rename the temporary file using the Rename File SVC (subopcode >95) and for a name specify the input file name. This will delete the input file, replacing it with the temporary file, using the same name as the deleted input file.
- Release the LUNO.

This method bypasses copying the temporary file to the input file and does not require naming the temporary file. DX10 automatically deletes temporary files created in this way when the system is initially loaded, so the operator does not have to delete intermediate files that remain after a system crash or other problem.

11

Debugging a Program

11.1 GENERAL INFORMATION

Flaws in software are commonly called "bugs". The process of removing flaws from software is called debugging. Modern programming techniques can drastically reduce the number of bugs in a program; however, the bugs which remain tend to be subtle and hard to find. DX10 provides several levels of debugging support, as follows:

- Several System Command Interpreter (SCI) commands provide debugging capabilities without requiring a special mode of operation.
- A special mode of operation allows a single task to be examined in detail during the execution process.

Since all of the debug commands interact with the terminal, special care must be taken when debugging a program that uses the terminal, because two processes requesting terminal support can be confusing to you. If the program being debugged requests use of a terminal, two terminals should be used: one for executing the program and one for debugging.

11.2 MODES OF DEBUGGING

There are two sets of debug commands:

- Commands used for debugging all tasks.
- Controlled task commands used for tasks that have been put into the debug mode through the use of the Execute Debug (XD) command.

NOTE

Putting a task into debug mode affects the execution of all debug commands as follows:

- Symbolic expressions may be used in place of integer expressions as responses to commands involving a controlled task.
- Every command functions as if the controlled task is unconditionally suspended.
- Every command leaves the controlled task unconditionally suspended.
- Tasks which unconditionally suspend themselves can be momentarily reactivated by some of the debug commands.
- The Command key automatically suspends the controlled task when executing the Proceed from Breakpoint (PB), Delete and Proceed from Breakpoint (DPB), or Resume Task (RT) commands.

11.2.1 Unconditional Suspend

Most of the debugging commands require that the task being debugged be unconditionally suspended either before or during the debug command. The "unconditional suspend" task state under DX10 (task state 6) is the state in which the task is dormant until activated by a command. There are several ways for a task to become unconditionally suspended:

- The task is bid with the suspend option selected. Either a supervisor call, the Execute and Halt Task (XHT) command, or the .DBID SCI primitive suspend a task when the task is bid.

  The XHT command is used for tasks normally executed by an Execute Task (XT) command. XHT places the task in a suspended state for debugging and displays the run ID of the task to the user. If you want execute and halt the task, and simultaneously place it in controlled mode, the Execute Debug (XD) command may be used with no input for the RUN ID prompt. The XD command performs the XHT and saves the run ID as the default for the Debugger commands.

  Use the .DBID primitive for tasks that interface through SCI, such as command processors which are normally bid using the .BID and .QBID primitives, described in Section 6. When the .DBID primitive is executed through SCI, the task is bid and immediately placed in a suspended state. The run ID of the task is saved in the synonym $$BT or it may be obtained by issuing a Show Task Status (STS) command.

- The task suspends itself.
- The task executes a breakpoint (XOP 15,15).
- The task is suspended by the SCI debug commands.

NOTE

A global LUNO must be assigned to the program file from which the task is bid.

Once the task has been placed in a suspended state, the Debugger may be used to assign breakpoints, simulate execution, display memory, and perform other debugging functions. When the debugging session is over, the task may be terminated by the Kill Task (KT) command. If the task was put into controlled mode by an XD command, it may be killed by responding YES to the KILL TASK? prompt of the Quit Debug (QD) command.

11.2.2 Symbols

The debug support provided allows symbolic debugging; whereby, you can specify labels within the task being debugged rather than memory addresses. This method of debugging is both convenient and meaningful since the source code list can be used as reference for the symbolic labels used. Symbolic constants consist of the link edit phase name, a period (.), the module identifier name (IDT), a period (.), and the symbol, an assembly language label. The syntax is defined as:

phase name.IDT name.symbol

NOTE

To have full symbolic capability, both the assembler and Link Editor must have used the SYMT option.

If the assembler did not use the SYMT option, but the Link Editor did, then symbols of the following form are available:

phase name.IDT name

If either the phase name or the IDT name of a symbol is omitted, the immediately preceding phase name or IDT name is used. The syntax is as follows:

.IDT name.symbol	(no phase name)
phase name..symbol	(no IDT name)
..symbol	(no phase or IDT name)

Examples:

PHASE1.MOD1.XYZ	References Phase = PHASE1 IDT = MOD1 Label = XYZ
.MOD2.MNO	References Phase = PHASE1 IDT = MOD2 Label = MNO
..ABC	References Phase = PHASE1 IDT = MOD2 Label = ABC

Four words of memory per symbol are required to store symbol values.

If the task being debugged is a single routine installed without being linked, then the symbolic constant consists of a period (.), the characters of the module identifier name, a period (.), and the characters of the symbol, as follows:

.IDT name.symbol

NOTE

Symbols may only be used for commands affecting a task that has been placed in the debug mode by the Execute Debug (XD) command.

Symbol encoding uses a hashing method which sometimes produces a seeming duplication of values for a symbol. In such cases, use another symbol.

11.2.3 Expressions

Constants (and symbolic constants for tasks in the debug mode) may be combined using the operators +, -, *, /, ( ), and < > to form expressions which may be used as command operands. The operators have the following meanings:

Operator	Meaning
+	Unary plus or addition
−	Unary minus or subtraction
*	Multiplication
/	Division
( )	Evaluation order
< >	Indicated memory location contents

NOTE

The right angle bracket, >, is regarded as a hexadecimal number indicator rather than the right part of < > whenever there are hexadecimal digits immediately following. Thus, no conflict arises.

Expressions are evaluated according to the following rules:

- Subexpressions delimited by ( ) and < > are evaluated first with the innermost expression evaluated before any other levels.
- Unless otherwise instructed by parentheses or angle brackets, unary + and - are evaluated first, multiplication and division are evaluated second, and addition and subtraction last.
- For operators at the same level, evaluation proceeds left to right.
- Arithmetic treats all constants as unsigned numbers.

For example, if .IDTNAM.BEGIN is memory address >7A, and if memory address >7F contains >3B, then the expression FF/(IDTNAM.BEGIN + 5 + -2 + 3 x > F) is evaluated as follows:

```
>FF/(<.IDTNAM.BEGIN+5> + -2+3x>F)
>FF/(<>7A+5> + -2+3x>F)
>FF/(<>7F> + -2+3x>F)
>FF/(>3B+ -2+>2D)
>FF/(>3B+(-2)+>2D)
>FF/(>39+>2D)
>FF/>66
2
```

These symbols may be used in expression lists in the same way as constants or symbolic constants. For example,

```
#PC + NAME.IDT - #R15
```

is a valid expression.

Several special symbols are allowed in expressions. These special symbols are:

Symbol	Description
#PC	Contents of the Program Counter
#WP	Contents of the Workspace Pointer
#ST	Contents of the Status Register
#Rn	Contents of the Workspace Register whose number corresponds to the number (0 through 15) given for n.

Character strings are also allowed in expressions. A character string is of the form 'XXXX_' where X is any valid ASCII character. The apostrophe can be represented in a character string by using double apostrophes. A character string may be any length, but only the leftmost four characters are significant. Strings shorter than four characters are right-justified with leading zeros. The value of a character string is an expression in the ASCII hexadecimal representation of the characters expressed as a 32-bit number.

String	Value
'ABCD'	41424344
'A'	00000041
'ABCDE'	41424244
' '	00000020
'A''B'	00412742

11.3 COMMANDS FOR ALL TASKS

The SCI commands described in the following paragraphs may be used for all tasks. These commands are most frequently used in debugging; however, they may be used whenever SCI is active. Many of the debug commands require the run-time task ID returned by the XT or XHT commands. Make note of the run-time task ID when the task is placed in execution. The Show Task Status (STS) command may be used to identify the run-time ID (which identifies the task to DX10).

Table 11-1 lists the paragraphs associated with each command for easier referencing.

Table 11-1. Debug Commands

Acronym	Debug Command	Paragraph Number
AB	Assign Breakpoints	11.3.3.1
ASB	Assign Simulated Breakpoints	11.3.6.1
AT	Activate Task	11.3.4.1
DB	Delete Breakpoints	11.3.3.2
DPB	Delete and Proceed from Breakpoint	11.3.3.3
DSB	Delete Simulated Breakpoints	11.3.6.2
FB	Find Byte	11.3.5.1
FW	Find Word	11.3.5.2
HT	Halt Task	11.3.4.2
LB	List Breakpoints	11.3.1.1
LLR	List Logical Record	11.3.1.2
LM	List Memory	11.3.1.3
LSB	List Simulated Breakpoints	11.3.6.3
LSM	List System Memory	11.3.1.4
MAD	Modify Absolute Disk	11.3.2.1
MADU	Modify Allocatable Disk Unit	11.3.2.2
MIR	Modify Internal Registers	11.3.2.3
MM	Modify Memory	11.3.2.4
MPI	Modify Program Image	11.3.2.5
MRF	Modify Relative to File	11.3.2.6
MSM	Modify System Memory	11.3.2.7
MWR	Modify Workspace Registers	11.3.2.8
PB	Proceed from Breakpoint	11.3.3.4
QD	Quit Debug Mode	11.3.6.4
RST	Resume Simulated Task	11.3.6.5
RT	Resume Task	11.3.4.3
SAD	Show Absolute Disk	11.3.1.5
SADU	Show Allocatable Disk Unit	11.3.1.6
SIR	Show Internal Registers	11.3.1.7
SP	Show Panel	11.3.1.8
SPI	Show Program Image	11.3.1.9
SRF	Show Relative to File	11.3.1.10
ST	Simulate Task	11.3.6.6
SV	Show Value	11.3.1.11
SWR	Show Workspace Registers	11.3.1.12
XD	Execute in Debug Mode	11.3.4.4
XHT	Execute and Halt Task	11.3.4.5

11.3.1 Data Display Commands

These SCI commands display the contents of memory, registers, and specified breakpoint addresses.

The LLR, LM, LSM, SAD, SADU, SP, SPI, and SRF commands have a similar format in their output. The following example uses the output of the SAD command to describe the format:

```
TRACK 0000   SECTOR 00  RECORD LENGTH 0120 BYTES (01 SECTORS).
003C  2020 0000 001A 0086 0003 07BF 0027 5324    .. .. .. .. .. .. .' S$
004C  5052 4F47 4120 2020 2020 2020 2020 0000   PR OG A                ..
005C  5324 4F56 4C59 4120 2020 2020 2020 2020   S$ OV LY A
006C  0000 2020 2020 2020 2020 5324 4C4F 4144   ..                S$ LO AD
007C  4552 0001 2020 2020 2020 2020 0000 0000   ER ..                .. ..
008C  0001 0001 2020 2020 2020 2020 2020 2020   .. ..
009C  2020 2020 0000 0001 0003 0000 0000 0000         .. .. .. .. .. ..
  SAME
0102  0000                                      ..
```

The first line of the output is a header line. (Some commands have a different format for header material.) All numbers are in hexadecimal. The memory is written in 9 columns of 4-digit numbers. The left-hand column contains the address of the first word in each line. The remainder of the line contains 8 columns of 4-digit hexadecimal numbers followed on the right by the text representation of the data on that line.

Each of the eight 4-digit numbers to the right of the first 4-digit number is said to be a *word* of data (also called a *data word*). The text on the right is divided into groups of two bytes, one such group for each hexadecimal data word. The bytes that contain values that correspond to printable ASCII characters are translated and printed as ASCII characters; those not corresponding to ASCII characters are printed as periods.

The word SAME in the example indicates that the last word printed on the preceding line is the same as all following words until the end of record is reached or until a different word is encountered.

11.3.1.1 List Breakpoints — LB. The LB command is used for displaying the breakpoints for a specified task. If the breakpoints are to be displayed for a system task, you must have a privileged user ID.

Prompts:

LIST BREAKPOINTS
RUN ID: integer (*)

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.1.2 List Logical Record — LLR. The LLR command lists the contents of a record or records in a file. The contents of the record or records specified are listed in both hexadecimal and ASCII representation. The amount displayed per record is a maximum of decimal 512 (hexadecimal 200) or the logical record length of the file, whichever is less.

Prompts:

LIST LOGICAL RECORD

PATHNAME:	pathname@	(*)
STARTING RECORD:	integer	(0)
NUMBER OF RECORDS:	[integer]	
LISTING ACCESS NAME:	[pathname@]	
MAXIMUM RECORD LENGTH:	[integer]	

Prompt Details:

PATHNAME:
The pathname that identifies the file in which the records to be listed reside.

STARTING RECORD:
A decimal or hexadecimal integer that identifies the first record that has contents to be listed.

NUMBER OF RECORDS:
A decimal or hexadecimal integer that identifies how many records are to be listed. A null response specifies that all records are to be listed.

LISTING ACCESS NAME:
The device name of a device or the pathname of a file to which the LLR command should write the contents of the record(s) specified. The default value is the terminal local file.

MAXIMUM RECORD LENGTH:
The size in bytes of the largest record you anticipate in the file or device being listed. The default value is 512 bytes.

11.3.1.3 List Memory — LM. The LM command is used to list the specified memory area of a task to a specified output device or file. If the task is not unconditionally suspended, it is temporarily suspended while the listing is being formatted.

Prompts:

LIST MEMORY

RUN ID:	integer	(*)
STARTING ADDRESS:	full exp	
NUMBER OF BYTES:	[full exp]	
LISTING ACCESS NAME:	[pathname@]	

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

STARTING ADDRESS:
The integer value that is the starting address of the memory area to be listed.

NUMBER OF BYTES:
The integer value that is the number of bytes of memory to be listed, beginning with the specified starting address. The default value is 16 bytes.

LISTING ACCESS NAME:
The device name or file name of the device or file where the memory list is to be output. The default value is the terminal local file.

11.3.1.4 List System Memory — LSM. The LSM command is used to list the memory occupied by the DX10 operating system. This command is similar to the List Memory (LM) command, except the user specifies an overlay name or ID instead of a run ID.

The LSM command is intended for use only by someone very familiar with DX10 source code.

Prompts:

```
LIST SYSTEM MEMORY
               OVERLAY ID:  {integer/alphanumeric}
         STARTING ADDRESS:  integer
          NUMBER OF BYTES:  integer             (040)
      LISTING ACCESS NAME:  [pathname@]
```

Prompt Details:

OVERLAY ID:
The overlay name or integer value specified in the Install Overlay (IO) command that is the ID of the overlay that has memory to be listed. By executing the Map Program File (MPF) command on the kernel program file whose name is specified at system generation, the user can inspect the acceptable overlay names and associated IDs.

STARTING ADDRESS:
The integer expression which is the starting address of the memory area to be listed.

NUMBER OF BYTES:
The integer value that is the number of bytes of memory to be listed, beginning with the specified starting address. The initial value is >40.

LISTING ACCESS NAME:
The device name or file name where the memory list is to be output. The default value is the terminal local file.

11.3.1.5 Show Absolute Disk — SAD. The SAD command displays the contents of a specified sector on a disk. The contents of 16 bytes are printed per line, with the address of the first byte printed as the first entry on the line. The contents of each pair of bytes are shown as four hexadecimal digits. At the right end of the line, the contents are printed as ASCII characters. The bytes that contain values that correspond to printable ASCII characters are translated and printed as ASCII characters; nonprinting ASCII characters are printed as periods.

The SAD command provides you with the option of interactively designating sectors, which allows you to cross sector boundaries and view a different sector without terminating the task.

Prompts:

SHOW ABSOLUTE DISK

DISK UNIT:	devicename@	(*)
INTERACTIVE?:	YES/NO	(NO)

If you answer NO to the INTERACTIVE? prompt, the following prompts appear:

TRACK:	integer exp	(*)
SECTOR:	integer exp	(*)
FIRST WORD:	integer exp	(0)
NUMBER OF WORDS:	[integer exp]	
OUTPUT ACCESS NAME:	[pathname@]	

Prompt Details:

DISK UNIT:

The device name assigned to the disk during system generation. Normally, the characters DS01 are used for the system disk and DSxx for other disks on the system; where xx is a two-digit decimal number greater than one (e.g., DS02).

INTERACTIVE?:

If you answer NO to this prompt, simply respond to the prompts that appear; the sector you specified, or a portion of that sector will appear.

If you answer YES to this prompt, it is assumed that you are working at a video display terminal (VDT). Sector 0 appears on your screen, and a prompt requesting the track and address of the sector you want to view appears at the upper right corner of the screen. Specify the track and sector numbers in hexadecimal form. If the sector that you specify exists, it appears on the screen. If it does not exist, an error code appears in the form ERR: 001C.

The functions of the F1 and F2 keys depend on your response to the INTERACTIVE? prompt. If you enter YES to this prompt, the F1 key scrolls forward to the next record, and the F2 key scrolls backward to the previous record. If you enter NO to this prompt, the F1 key scrolls forward to the next page within the same record, and F2 scrolls backward to the previous page within the same record.

The F3 key allows you to specify a sector in the following three ways:

- When you specify a sector by track number and sector number, the following prompts appear:

  TRACK: 0000 SECTOR: 0000

- When you specify a sector by head number, cylinder number, and sector number, the following prompts appear:

  HEAD: 0000 CYL: 0000 SECTOR: 0000

- When you specify a sector by allocatable disk unit (ADU) number and sector offset number, the following prompts appear:

  ADU: 0000 SECTOR OFFSET: 0000

TRACK:
: The integer value that is the starting track address from which to begin printing the contents of the disk.

SECTOR:
: The integer value that is the starting sector address, within the specified disk track, from which to begin printing the contents of the disk.

FIRST WORD:
: The integer value that is the word offset, within the specified disk sector, from which to begin printing the contents of the disk.

NUMBER OF WORDS:
: The integer value that is the number of words of the specified sector to print. The default value is the disk sector size.

OUTPUT ACCESS NAME:
: The device name or file name of a device or file where the contents of the specified absolute disk address is to be printed. The default value is the terminal local file.

11.3.1.6 Show Allocatable Disk Unit — SADU. The SADU command is used to output the contents of the specified allocatable disk units (ADUs) to the specified device.

All disks on a DX10 system are addressed in ADUs, the basic addressable disk unit in a DX10 system. The maximum number of ADUs on a disk is 65,535. Therefore, if a disk contains more than 65,535 sectors, multiple sectors are used as ADUs.

Prompts:

SHOW ALLOCATABLE DISK UNIT

DISK UNIT:	devicename@	
ADU NUMBER:	integer exp	
SECTOR OFFSET:	integer exp	
FIRST WORD:	integer	(0)
NUMBER OF WORDS:	[integer exp]	
OUTPUT ACCESS NAME:	[pathname@]	(*)

Prompt Details:

DISK UNIT:
The device name assigned to the disk during system generation. Normally, the characters DS01 are used for the system disk and DSxx for other disks on the system, where xx is a two-digit decimal number greater than one (e.g., DS02).

ADU NUMBER:
The integer value that is the ADU with contents to be listed.

SECTOR OFFSET:
The integer value that is the sector of the ADU with contents to be listed.

FIRST WORD:
The integer value that is the word offset, within the specified sector, from which to begin listing the contents of the ADU.

NUMBER OF WORDS:
The integer value that is the number of words of the specified sector to list. The default value is the disk ADU size.

OUTPUT ACCESS NAME:
The device or file name where the contents of the specified ADU are to be listed. The default value is the terminal local file.

11.3.1.7 Show Internal Registers — SIR. The SIR command is used to display the task state and the contents of the internal registers of a task: program counter (PC), workspace pointer (WP), workspace register (WR), status register (ST), memory, and breakpoints. The STATE field is the state of the task before it was suspended to show the contents of the internal registers. The remainder of the display reflects the internal register values in effect after the task was suspended.

The character string representation of the status register follows the hexadecimal value and may include the following characters:

L = logical greater than
A = arithmetic greater than
E = equal
C = carry
O = overflow
P = parity
X = XOP in progress
S = privileged mode
M = map file

If the internal registers are to be shown for a system task, the user must have a privileged user ID.

Prompts:

```
SHOW INTERNAL REGISTERS
                        RUN ID:  integer  (*)
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.1.8 Show Panel — SP. The SP command is used to display the debug panel for a specified task. If the task is not unconditionally suspended, it will be temporarily suspended while the panel is being formatted and displayed. The displayed task state is the state of the task before it was suspended. The debug panel consists of the following:

- Internal registers
- Workspace registers
- Breakpoints
- Memory display
- Task state

The SP command also shows the character string representation of the status register.

If the debug panel to be displayed is for a system task, the user must have a privileged user ID.

Prompts:

```
SHOW PANEL
                    RUN ID:  integer          (*)
            MEMORY ADDRESS:  [full exp]
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs are obtained by executing the Show Task Status (STS) command.

MEMORY ADDRESS:
The integer value that is the starting memory address for the memory portion of the debug panel display. The default value is the current PC address.

11.3.1.9 Show Program Image — SPI. The SPI command is used to display the disk-resident memory image of a module (defined as a task, procedure, or overlay) for a specified program.

Prompts:

```
SHOW PROGRAM IMAGE
              PROGRAM FILE:  filename@                   (*)
        OUTPUT ACCESS NAME:  [pathname@]                 (*)
               MODULE TYPE:  {T/P/O}                     (*)
        MODULE NAME OR ID:   {alphanumeric/integer}      (*)
                   ADDRESS:  integer                     (*)
                    LENGTH:  integer                     (040)
```

Prompt Details:

PROGRAM FILE:
The file name of or the LUNO assigned to the program file on which the program (task, procedure, or overlay) has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the SPI command. If zero is specified, the .S$PROGA program file is assumed.

OUTPUT ACCESS NAME:
The device name or file name where the display of the memory image of the program is to be written. The default value is the terminal local file.

MODULE TYPE:
The type of program with a memory image to be displayed. The following characters are valid responses:

T = Task
P = Procedure
O = Overlay

MODULE NAME OR ID:
The characters or the associated ID that identifies the program on the specified program file.

ADDRESS:
The integer value that is the starting address of the memory image to be displayed.

LENGTH:
The integer value that is the number of words of the memory image to be displayed.

11.3.1.10 Show Relative to File — SRF. The SRF command displays any record or portion of a record within a file. It assumes that you have knowledge of the file structure and allows you to address any word within the file. The SRF command provides you with the option of interactively designating records, which allows you to cross record boundaries and view a different record without terminating the task.

Prompts:

```
SHOW RELATIVE TO FILE
                    PATHNAME:  filename@        (*)
                INTERACTIVE?:  YES/NO
```

If you answer NO to the INTERACTIVE? prompt, the following prompts appear:

```
              RECORD NUMBER:  integer           (*)
                 FIRST WORD:  integer           (*)
            NUMBER OF WORDS:  [integer]
         OUTPUT ACCESS NAME:  [pathname@]
```

Prompt Details:

PATHNAME:
The name of the file with a record to be displayed.

INTERACTIVE?:
If you answer NO to this prompt, simply respond to the prompts that appear; the record you specified, or a portion of that record, will appear.

If you answer YES to this prompt, it is assumed that you are working at a video display terminal (VDT). Record 0 appears on your screen, and a prompt requesting the number of the record you want to view appears at the upper right corner of the screen. Specify the record number using its hexadecimal value. If the record that you specify exists, it appears on the screen. If it does not exist, an error code appears in the form ERR: 0030.

The functions of the F1 and F2 keys depend on your response to the INTERACTIVE? prompt. If you enter YES to this prompt, the F1 key scrolls forward to the next record, and the F2 key scrolls backward to the previous record. If you enter NO to this prompt, the F1 key scrolls forward to the next page within the same record, and F2 scrolls backward to the previous page within the same record.

The Previous Line and Next Line keys perform the same function with either a YES or NO response to this prompt. If the entire record will not fit on the screen, you can advance the data, line by line, by pressing the Previous Line key. The Next Line key moves the data backward through the data window.

Interactive use of SRF also includes a hash function. Press the F4 key to help locate a file descriptor record (FDR) within a directory.

When you press the F4 key, the following prompt appears:

HASH - NUMBER OF DIRECTORY ENTRIES:

Respond with the number of directory entries (in hexadecimal) for the directory that you are using, and press the Return key. The following prompt appears on your screen:

FILE NAME TO HASH:

Enter the file name of the FDR that you want to locate, then press the Return key.

The system hashes the file name to a record number and that record appears on the screen. Since several files can hash to the same directory entry, the FDR may not appear on this record. Use the F1 key to search sequentially for the desired FDR.

RECORD NUMBER:
The integer value that is the physical record number within the file of the first word of data to be displayed.

FIRST WORD:
The integer value that is the byte offset within the record to be displayed.

NUMBER OF WORDS:
The integer value that is the number of words of the record to be displayed. If the number of words specified is larger than the number of words left in the record, words from succeeding records will be displayed. The default is to display the number of words required to reach the end of the physical record.

OUTPUT ACCESS NAME:
The pathname of a device or file where the results of the SRF command are to be listed. The default is the terminal local file.

11.3.1.11 Show Value — SV. The SV command is used to display the value of a specified expression. The hexadecimal, decimal, and ASCII representations of the value are given.

Prompts:

```
SHOW VALUE
                    EXPRESSION:   full exp
```

Prompt Details:

EXPRESSION:
The integer and/or character(s) expression with a value to be displayed. If a task is being debugged and is a controlled task, the expression can be symbolic.

11.3.1.12 Show Workspace Registers — SWR. The SWR command is used to display the current workspace of a task. If the task is not unconditionally suspended, it is temporarily suspended while the workspace is displayed.

If the terminal requesting the command is a VDT, the SWR command functions the same as the Show Panel (SP) command. If the workspace to be displayed is for a system task, the user must have a privileged user ID.

Prompts:

```
SHOW WORKSPACE REGISTERS
                          RUN ID:   integer   (*)
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.2 Data Modification Commands

These commands are used to place specified data on a disk or change data at an absolute word address. Modification of specified ADUs, internal registers, memory image, or programs may be accomplished using these debugging commands.

11.3.2.1 Modify Absolute Disk — MAD. The MAD command is used to place specified data on a disk at a specified absolute track, sector, and word address and may only be executed by privileged users. Data is entered in groups of word values to be placed on disk. Word values must be separated from each other with a comma and loaded on disk in successive addresses. The verification parameter allows you to enter a string of words to be compared to the data at the specified address. If there is not a correspondence between the string of words and the data at the specified address, the modification does not take place.

The MAD command provides you with the option of interactively designating and modifying sectors, which allows you to cross sector boundaries and view and/or modify a different sector without terminating the task. It also allows you to modify a disk in hexadecimal or ASCII format.

NOTE

Since the MAD command has the capability to write anything anywhere on the disk and can therefore destroy the DX10 system image, the verify option should always be used.

Prompts:

```
MODIFY ABSOLUTE DISK
                     DISK UNIT:  devicename@     (*)
                  INTERACTIVE?:  YES/NO          (NO)
```

If you answer NO to the INTERACTIVE? prompt, the following prompts appear:

OUTPUT ACCESS NAME:	[pathname@]	
TRACK:	integer exp	(*)
SECTOR:	integer exp	(*)
FIRST WORD:	integer exp	(*)
VERIFICATION DATA:	[integer exp list]	
DATA:	integer exp list	(*)

Prompt Details:

DISK UNIT:

The device name of the disk device assigned during system generation. Normally, the characters DS01 are used for the system disk and DS0x for other disks on the system, where x is a digit greater than one.

INTERACTIVE?:

If you answer NO to this prompt, simply respond to the prompts that appear; the sector you specified, or a portion of that sector will be modified.

If you answer YES to this prompt, it is assumed that you are working at a video display terminal (VDT). Track 0, Sector 0 appears on your screen and a prompt requesting the number of the track and sector you want to modify appears at the upper right corner of the screen. Enter the numbers in hexadecimal form. If the sector that you specify exists, it appears on the screen. If it does not exist, an error code appears in the form ERR: 001C.

The functions of the F1 and F2 keys depend on your response to the INTERACTIVE? prompt. If you enter YES to this prompt, the F1 key scrolls forward to the next record, and the F2 key scrolls backward to the previous record. If you enter NO to this prompt, the F1 key scrolls forward to the next page within the same record, and F2 scrolls backward to the previous page within the same record.

While you are in the interactive mode, data is displayed in hexadecimal format on the left side of the screen and in ASCII on the right side of the screen. When the hexadecimal data cannot be represented as an ASCII character, a dot is displayed. To begin editing while you are in the interactive mode, press the F8 key. The cursor will move to the upper left portion of the hexadecimal data. You can edit the data in either hexadecimal or ASCII format. To move from the hexadecimal data on the left side of the screen to the ASCII data on the right side of the screen, press the F7 key. To move from the ASCII data to the hexadecimal data, press the F6 key.

You can move the cursor forward through the data fields one at a time by pressing the Return key. This advances the cursor to the next field that you can edit. The Previous Field key moves the cursor backward to the first field that you can edit. The Previous Line key moves the cursor up one line (if the line is available), and the Next Line key moves the cursor down one line (if the line is available). If the cursor cannot move where you direct it to move, it remains in the current field.

As you edit the hexadecimal data on the left side of the screen, the ASCII data on the right of the screen is modified.

The data that you enter is not written to the disk until you press the Enter key. MAD then returns to the show mode on the same sector that you were editing. If you want to abort the update to the sector, press the Command key; this returns you to the show mode. The F1 and F2 keys also abort the changes you have made before they move the cursor to the appropriate sector in the show mode.

The F3 key allows you to specify the sector in the following three ways:

- When you specify a sector by track number and sector number, the following prompts appear:

  TRACK: 0000   SECTOR: 0000

- When you specify a sector by head number, cylinder number, and sector number, the following prompts appear:

  HEAD: 0000   CYL: 0000   SECTOR: 0000

- When you specify a sector by allocatable disk unit (ADU) number and sector offset number, the following prompts appear:

  ADU: 0000   SECTOR OFFSET: 0000

OUTPUT ACCESS NAME:
The device or file name where the contents of the specified absolute disk address are to be printed. The default value is the terminal local file.

TRACK:
The integer value which is the starting track address from which to begin the disk modification.

SECTOR:
The integer value that is the starting sector address, within the specified disk track, from which to begin the disk modification.

FIRST WORD:
The integer value that is the starting word address, within the specified disk sector, from which to begin the disk modification.

VERIFICATION DATA:
If specified, the integer value contained in the specified starting address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive words, beginning with the specified first word. This feature verifies that the data you are modifying has the value(s) you expect and/or that the data you are entering has the correct checksum. If a bad comparison results, the modification does not take place.

DATA:

The integer value to replace the existing value contained in the specified first word. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive words, beginning with the first word.

11.3.2.2 Modify Allocatable Disk Unit — MADU. The MADU command is used to modify a specified allocatable disk unit (ADU). If verification data does not match the data already on the disk, modification will not be performed.

All disks on a DX10 system are addressed in ADUs. The maximum number of ADUs on a disk is 65,535. Therefore, if a disk contains more than 65,535 sectors, multiple sectors are used as ADUs. ADUs are the basic addressable disk unit in a DX10 system.

Prompts:

```
MODIFY ALLOCATABLE DISK UNIT
                     DISK UNIT:  devicename@
            OUTPUT ACCESS NAME:  [pathname@]   (*)
                    ADU NUMBER:  integer exp
                 SECTOR OFFSET:  integer exp
                    FIRST WORD:  integer exp
             VERIFICATION DATA:  [integer exp list]
                          DATA:  integer exp list
```

Prompt Details:

DISK UNIT:

The device name of the disk assigned during system generation. Normally, the characters DS01 are used for the system disk and DS0x for other disks on the system where x is a digit greater than one.

OUTPUT ACCESS NAME:

The device or file name where the results of the ADU modification is to be listed. The default value is the terminal local file.

ADU NUMBER:

The integer value that is the ADU with contents to be modified.

SECTOR OFFSET:

The integer value that is the sector of the ADU with contents to be modified.

FIRST WORD:

The integer value that is the starting word offset, within the specified sector, where modifications of the ADU are to begin.

VERIFICATION DATA:

If specified, the integer value contained in the specified first word address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive words, beginning with the specified first word.

DATA:

The integer value to replace the existing value contained in the specified first word. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive words, beginning with the first word.

11.3.2.3 Modify Internal Registers — MIR. The MIR command is used to modify the internal registers of a task: program counter (PC), workspace pointer (WP), and status register (ST). If the task being debugged is not a privileged task, then only bits 0 through 6 of the status register can be modified with this command. If the task is not unconditionally suspended, it is temporarily suspended while the command is interacting with the register modification.

The Modify Internal Registers (MIR) command is interactive. The Return key can be pressed after the register and its contents have been displayed and/or modified to cause the next register and its contents to be displayed. Also, by pressing the Command key, SCI is returned to command mode.

If the internal registers to be modified are for a system task, the user must have a privileged user ID.

Prompts:

```
MODIFY INTERNAL REGISTERS
                         RUN ID:  integer  (*)
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

11.3.2.4 Modify Memory — MM. As in the MIR command, the Modify Memory (MM) command is interactive. The MM command is used to modify the memory image of a task, starting at the address specified. If the task is not unconditionally suspended, it is temporarily suspended while the command is interacting. Swapping does not affect the modification process. Consecutive memory addresses and their values can be displayed and/or modified by pressing the Return key. Pressing the Command key will return SCI to command mode.

Prompts:

MODIFY MEMORY

RUN ID:	integer	(*)
ADDRESS:	full exp	

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

ADDRESS:
The integer value of the first memory address to be modified.

11.3.2.5 Modify Program Image — MPI. The MPI command is used to modify a program (defined to be a task, procedure, or overlay) in a specified program file.

Prompts:

MODIFY PROGRAM IMAGE

PROGRAM FILE:	filename@	(*)
OUTPUT ACCESS NAME:	[pathname@]	(*)
MODULE TYPE:	{T/P/O}	(*)
MODULE NAME OR ID:	{alphanumeric/integer}	(*)
ADDRESS:	integer	(*)
VERIFICATION DATA:	[integer(s)]	
DATA:	integer(s)	
CHECKSUM:	[integer(s)]	
RELOCATION OF DATA?:	[YES/NO...YES/NO]	

Prompt Details:

PROGRAM FILE:
The file name of or the LUNO assigned to the program file on which the program (task, procedure, or overlay) to be modified has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the MPI command. If zero is specified, the .S$PROGA program file is assumed.

OUTPUT ACCESS NAME:
The device name or file name where the results of the memory image modification of the program are to be written. If a null response is specified, the terminal local file is used.

MODULE TYPE:
The type of program with a memory image to be modified. The following characters are valid responses:

T = Task
P = Procedure
O = Overlay

MODULE NAME OR ID:
The character(s) or the associated ID which identifies the program on the specified program file.

ADDRESS:
The integer value which is the starting address of the memory image to be modified.

VERIFICATION DATA:
If specified, the integer value contained in the specified starting address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in consecutive memory addresses, beginning with the specified starting address.

DATA:
The integer value to replace the existing value contained in the specified starting address. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in consecutive memory addresses, beginning with the specified starting address.

CHECKSUM:
The checksum is an exclusive OR of each word of new data. If the checksum is not known and a null response is entered, the checksum is printed to the device or file specified in response to the OUTPUT ACCESS NAME prompt.

RELOCATION OF DATA?:
If YES is specified, the data value is relocated when the task is loaded into memory for execution. NO specifies that the data value should not be relocated. If a list of data values are specified in response to the DATA prompt and relocation is desired, the user must specify which values are to be relocated. That is, a YES or NO response must be entered for each corresponding data value. If there is a list of YES or NO responses, they must be separated by commas.

11.3.2.6 Modify Relative to File — MRF. The MRF command provides you with the option of interactively designating and modifying records, which allows you to cross record boundaries and view and/or modify a different record without terminating the task. It also allows you to modify a file in hexadecimal or ASCII format.

In the noninteractive mode, the MRF command optionally verifies that the data you are modifying has the value(s) you expect and/or that the data you are entering has the correct checksum. You should use the verification features whenever possible.

Prompts:

MODIFY RELATIVE TO FILE		
PATHNAME:	filename@	(*)
INTERACTIVE?:	YES/NO	

If you answer NO to the INTERACTIVE? prompt, the following prompts appear:

OUTPUT ACCESS NAME:	[pathname@]	
RECORD NUMBER:	integer	(*)
FIRST WORD:	integer	(*)
VERIFICATION DATA:	[integer exp list]	
DATA:	integer exp list	(*)
CHECKSUM:	[integer]	

Prompt Details:

PATHNAME:
The file name with contents to be modified.

INTERACTIVE?:
If you answer NO to this prompt, simply respond to the prompts that appear; the record you specified, or a portion of that record, will be modified.

If you answer YES to this prompt, it is assumed that you are working at a video display terminal (VDT). Record 0 appears on your screen and a prompt requesting the number of the record you want to modify appears at the upper right corner of the screen. Enter the record number using its hexadecimal value. If the record that you specify exists, it appears on the screen. If it does not exist, an error code appears in the form ERR: 0030.

Your response to the INTERACTIVE? prompt determines the functions of the F1 and F2 keys on your terminal. If you enter YES in response to this prompt, the F1 key scrolls forward to the next record, and the F2 key scrolls backward to the previous record. If you respond NO to this prompt, the F1 key scrolls forward to the next page within the same record, and F2 scrolls backward to the previous page within the same record.

While you are in the interactive mode, data is displayed in hexadecimal format on the left side of the screen and in ASCII format on the right side of the screen. When the hexadecimal data cannot be represented as an ASCII character, a dot is displayed. To begin editing while you are in the interactive mode, press the F8 key. The cursor will move to the upper left portion of the hexadecimal data. You can edit the data in either hexadecimal or ASCII format. To move from the hexadecimal data on the left side of the screen to the ASCII data on the right side of the screen, press the F7 key. To move from the ASCII data to the hexadecimal data, press the F6 key.

You can move the cursor forward through the data fields, one at a time, by pressing the Return key. This advances the cursor to the next field that you can edit. The Previous Field key moves the cursor backward to the first field that you can edit. The Previous Line key moves the cursor up one line (if the line is available), and the Next Line key moves the cursor down one line (if the line is available). If the cursor cannot move where you direct it to move, it remains in the current field.

11.3.2.7 Modify System Memory — MSM.

CAUTION

This command is available for use only by TI representatives.

The MSM command is used to modify the memory occupied by the DX10 operating system. It is similar to the Modify Memory (MM) command, except that an overlay name or ID is specified instead of a task run ID. Consecutive memory addresses can be displayed and/or modified by pressing the Return key. Pressing the Command key returns SCI to command mode.

Prompts:

```
MODIFY SYSTEM MEMORY
                          OVERLAY ID:   {integer/alphanumeric}
                             ADDRESS:   integer
```

Prompt Details:

OVERLAY ID:
The overlay name or integer value that is the ID of the overlay installed on the kernel program file whose name is specified at system generation, with a memory to be modified. By executing the Map Program File (MPF) command on the kernel program file, the user can inspect the acceptable overlay names and associated IDs.

ADDRESS:
The integer value of the memory address at which to begin modifying memory.

11.3.2.8 Modify Workspace Registers — MWR. The MWR command is used to modify specific workspace registers of a task. If the task is not unconditionally suspended, it is temporarily suspended while the command is modifying the registers.

As in the Modify Memory (MM) command, the MWR command is interactive; the Return key must be pressed to enter a new value for a workspace register. Also, pressing the Command key will terminate the MWR command and place SCI in command mode.

If the workspace registers to be modified are for a system task, the user must have a privileged user ID.

Prompts:

```
MODIFY WORKSPACE REGISTERS
                              RUN ID:   integer          (*)
                     REGISTER NUMBER:   integer exp      (0)
```

As you edit the hexadecimal data on the left side of your screen, the ASCII data on the right side of your screen is modified.

The data that you enter is not written to the file until you press the Enter key. MRF then returns to the show mode on the same record that you were editing. If you want to abort the update to the record, press the Command key; this returns you to the show mode. The F1 and F2 keys also abort the changes you have made before they move the cursor to the appropriate record in the show mode.

Interactive use of MRF also includes a hash function. Press the F4 key to help locate a file descriptor record (FDR) within a directory.

When you press the F4 key, the following prompt appears:

HASH - NUMBER OF DIRECTORY ENTRIES:

Respond with the number of directory entries (in hexadecimal) for the directory that you are using and press the Return key. The following prompt appears on your screen:

FILE NAME TO HASH:

Enter the file name of the FDR that you want to locate, then press the Return key.

The system hashes the file name to a record number and that record appears on the screen. Since several files can hash to the same directory entry, the FDR may not appear on this record. Use the F1 key to search sequentially for the desired FDR.

OUTPUT ACCESS NAME:
The device name or file name where the results of the MRF command are to be listed. If a null response is specified, the terminal local file is used.

RECORD NUMBER:
The integer value which is the physical record number within the file to be modified. If the specified word address is over 64K bytes, the user must supply the sector offset as the response to this prompt.

FIRST WORD:
The integer value which is the starting byte offset where the modification of the record is to begin. The byte offset must be on an even boundary.

VERIFICATION DATA:
If specified, the integer value contained in the specified first word address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive word addresses, beginning with the specified first word address. If the verification data is entered and does not match the actual data, MRF will not perform the modification.

DATA:

The integer value to replace the existing value contained in the specified first word address. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive word addresses, beginning with the specified first word address.

CHECKSUM:

The checksum is an exclusive OR of each word of new data. If the checksum is not known and a null response is entered, the checksum is printed to the device or file specified in response to the OUTPUT ACCESS NAME prompt.

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

REGISTER NUMBER:
The integer value that is the beginning workspace register to be displayed.

11.3.3 Breakpoint Commands

The purpose of setting breakpoints is to suspend the task at critical points in its execution in order to examine the memory contents of specified address(es).

11.3.3.1 Assign Breakpoints — AB. The AB command is used to replace the specified contents of the address(es) of a task with a breakpoint (an XOP 15,15 instruction >2FCF), which stops execution of the task at that location. Thus, a task may be suspended at any location in its execution.

The task is temporarily suspended, while the breakpoints are inserted, then restored to its original state. Unless the task is being simulated, the user must monitor the task with the Show Internal Registers (SIR) or Show Panel (SP) command to determine when it reaches a breakpoint. When the breakpoint occurs, the task is placed in task state 6 (unconditional suspend). To proceed, the Proceed from Breakpoint (PB), Delete and Proceed from Breakpoint (DPB), or the Delete Breakpoint (DB) and Resume Task (RT) commands must be executed.

The contents of the address(es) are saved and can be restored by the Delete Breakpoints (DB) command. A maximum number of breakpoints can be in effect within a job at any one time; an attempt to use more than this number of breakpoints generates an error message. If the run-time ID specifies a system task, the user must be a privileged user or the command is aborted. Breakpoints may not be assigned when working in the DX10 system area.

The DB, DPB, RT, PB, and ST commands are discussed later in this section.

Prompts:

```
ASSIGN BREAKPOINT
                    RUN ID:  integer  (*)
              ADDRESS(ES):  full exp list
```

Prompt Details:

RUN ID:
A valid run ID in the user's current job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

ADDRESS(ES):
The integer value(s) of the address(es) within the task where the breakpoint(s) are to occur. Addresses must be separated by a comma.

11.3.3.2 Delete Breakpoints — DB. The DB command is used to delete a breakpoint(s) from a specified address(es) in a task and restore the original value at the address(es) that was replaced with an XOP 15,15 instruction by the AB command. The task is temporarily suspended while the breakpoint(s) is deleted; then the original task state is restored. Deleting a breakpoint at which a task is stopped does not cause the task to resume execution; the Resume Task (RT) command must be used to cause the task to resume execution.

The parameters of the DB command are interpreted by the DB processor as in the AB command, with the following exceptions:

- If no address is specified in response to the ADDRESS(ES): prompt, the breakpoint at which the task is currently stopped is deleted.
- If ALL is specified in response to the ADDRESS(ES): prompt, all breakpoints for that task are deleted.
- If the breakpoint does not exist at the specified address, or a breakpoint within a list of breakpoints does not exist, the user is warned with an error message followed by a display of the debug panel to show the breakpoint status.

Prompts:

```
DELETE BREAKPOINTS
                          RUN ID:  integer  (*)
                     ADDRESS(ES):  [{full exp list/ALL/current breakpoint}]
```

Prompt Details:

RUN ID:
: A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

ADDRESS(ES):
: The integer value(s) of the breakpoint address(es) within the task set by the AB command. Addresses must be separated by a comma. If ALL is entered, all breakpoints are deleted. The default breakpoint is the one at which the task is currently stopped.

11.3.3.3 Delete and Proceed from Breakpoint — DPB. The DPB is used to proceed from a breakpoint at which a task is currently stopped and to delete that breakpoint. If the breakpoint has already been deleted, the command functions as if it were a Proceed from Breakpoint (PB) command.

Prompts:

```
DELETE AND PROCEED FROM BREAKPOINT
                          RUN ID:  integer  (*)
         DESTINATION ADDRESS(ES):  [full exp list]
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

DESTINATION ADDRESS(ES):
The integer value(s) of the address(es) within the task which are additional breakpoints to be set. A null response specifies that no new breakpoints are to be set.

11.3.3.4 Proceed from Breakpoint — PB. The PB command is used to resume execution of a task that is stopped at a breakpoint without deleting the breakpoint. The task is resumed, executing the instruction at the breakpoint at which it is currently stopped; however, the breakpoint remains active. If the task is not currently at a breakpoint, the user is notified by a warning message that the task is not at a breakpoint, and the task remains in whatever state it was in before the PB command.

The PB command may also be used to assign new breakpoints in the specified task by responding to the DESTINATION ADDRESS(ES) prompt. Breakpoints are set, if possible, at all the specified destination addresses. If no destination address(es) is specified, the task resumes execution but no breakpoints are set.

Prompts:

```
PROCEED FROM BREAKPOINT
                     RUN ID:  integer  (*)
     DESTINATION ADDRESS(ES):  [full exp list]
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

DESTINATION ADDRESS(ES):
The integer value(s) of the address(es) within the task where the new breakpoints are to occur. Addresses must be separated by a comma. The default value is no new breakpoints.

11.3.4 Task Control Commands

The control commands are used to unconditionally suspend and activate the task during the debugging process.

11.3.4.1 Activate Task — AT. The AT command is used to activate an unconditionally suspended task.

Prompts:

```
ACTIVATE TASK
                     RUN ID:  integer  (*)
```

Prompt Details:

RUN ID:
A valid task run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.4.2 Halt Task — HT. The HT command is used to unconditionally suspend a task at the end of the current time slice. If the task is already unconditionally suspended, it has no effect on the task. If the task is not in the active state, the HT command waits five seconds for the task to reach unconditional suspend, then gives the user the option of aborting or continuing to wait. This option occurs every five seconds if the task is not active and the HT command is executed.

If the task cannot be suspended, the following message is displayed:

UNABLE TO SUSPEND TASK. CURRENT STATE = XX. CONTINUE COMMAND?

If a YES response is entered, another attempt is made to suspend the task. If unsuccessful, the message is displayed again. A NO response to the preceding message causes the following message to be displayed:

DO YOU WISH TO LEAVE SUSPENSION PENDING?

A YES response leaves the suspension pending, while a NO response terminates the suspension attempt.

If the specified task is a system task, the user must have a privileged user ID.

Prompts:

HALT TASK
RUN ID: integer (*)

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.4.3 Resume Task — RT. The RT command is used to activate a task at the point at which it was suspended. The specified task must be unconditionally suspended when this command is executed or an error is indicated. The Delete Breakpoint (DB) and the RT command, Delete and Proceed from Breakpoint (DPB) or Proceed from Breakpoint (PB) commands must be used to restart a task halted at a breakpoint. The RT command should be used instead of the Activate Task (AT) command, to reactivate a task halted by the Halt Task (HT) command.

Prompts:

```
RESUME TASK
                        RUN ID:  integer   (*)
```

Prompt Details:

RUN ID:
The response to this prompt must be a valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

11.3.4.4 Execute in Debug Mode — XD. The XD command is used to place a specified task into controlled mode. The run-time ID is optional but cannot be the ID of a system task. If no run-time ID is specified, an automatic call is made to the Execute and Halt Task (XHT) command to place the task into execution.

The symbol table object file is optional and its presence determines whether symbolic expressions are allowed on any of the subsequent debug commands. If a symbol table was specified to the Link Editor (SYMT option was selected) and if the controlled task symbol table object file is specified, then symbolic expressions involving symbols in the object code symbol table may be used in commands that call for string parameters.

The debugger may be used to simulate 990 computer object code. The command defaults to the object code of the host computer.

Only one task for each station may be in debug mode at a given time.

Prompts:

```
EXECUTE IN DEBUG MODE
                      RUN ID:  [integer]        (*)
   SYMBOL TABLE OBJECT FILE:  [filename@]      (*)
        990/12 OBJECT CODE?:  YES/NO-          (YES)
```

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs are obtained by executing the Show Task Status (STS) command.

If a null response is specified, the prompts for the Execute and Halt Task (XHT) command are displayed. Refer to the XHT command for information concerning responses to these prompts.

SYMBOL TABLE OBJECT FILE:

The file name specified to the Link Editor if the SYMT option has been selected. By specifying this file name in response to the prompt, the user is allowed to use symbolic expressions which involve symbols in the object code symbol table on any debug command prompt which calls for a character(s) response. If a null response is entered, no symbol table file is used and symbolic expressions are not allowed.

990/12 OBJECT CODE?:

If YES is entered in response to this prompt, the debugger simulates 990/12 object code if executing on a 990/12 computer. If NO is entered, the debugger simulates 990/10 object code whether executing on a 990/10 or 990/12 computer.

11.3.4.5 Execute and Halt Task — XHT. The XHT command is used to place a task in memory in a suspended state so that it can be debugged. Typically, the user places the task to be debugged in memory using XHT, establishes the debug environment (including breakpoints), and then activates the task using the Resume Task (RT) command.

Prompts:

```
EXECUTE AND HALT TASK
        PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
             TASK NAME OR ID:  {alphanumeric/integer}   (*)
                       PARM1:  integer                  (0)
                       PARM2:  integer                  (0)
                  STATION ID:  {integer/ME}             (ME)
```

Prompt Details:

PROGRAM FILE OR LUNO:

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

TASK NAME OR ID:

The name or the associated installed ID of the task to be halted.

PARM1:

An integer value to be passed to the task being halted, determined by the programmer who wrote the task.

PARM2:

A second integer value to be passed to the task being halted, determined by the programmer who wrote the task.

STATION ID:

The station ID (e.g., 1, 2) with which the task is to be associated or the two-character pseudo device name of ME. If > FF is entered, the task is not associated with any station.

11.3.5 Search Commands

The search commands are used to search for the specified value(s) in a memory area of a task.

11.3.5.1 Find Byte — FB. The FB command is used to search for the specified value(s) in a memory area of a task; with the search beginning on a byte boundary. If the specified value is found, the corresponding memory address is displayed. If the task is not unconditionally suspended, it is temporarily suspended while the search is performed.

Prompts:

FIND BYTE

RUN ID:	integer (*)
VALUE(S):	full exp list
STARTING ADDRESS:	[full exp]
ENDING ADDRESS:	[full exp]

Prompt Details:

RUN ID:
: A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

VALUE(S):
: The integer value(s) to find in the memory area of the task.

STARTING ADDRESS:
: The integer value that is the starting address of the memory area to be searched. The default is zero.

ENDING ADDRESS:
: The integer value that is the ending address of the memory area to be searched. The default is end of task.

11.3.5.2 Find Word — FW. The FW command is used to search for the specified value(s) in a memory area of a task; with the search beginning on a word boundary. If the specified value is found, the corresponding memory address is displayed. If the task is not unconditionally suspended, it is temporally suspended while the search is performed.

Prompts:

FIND WORD

RUN ID:	integer (*)
VALUE(S):	full exp list
STARTING ADDRESS:	[full exp]
ENDING ADDRESS:	[full exp]

Prompt Details:

RUN ID:
A valid run ID in the user's job. Current run IDs can be obtained by executing the Show Task Status (STS) command.

VALUE(S):
The integer value(s) to find in the memory area of the task.

STARTING ADDRESS:
The integer value which is the starting address of the memory area to be searched. If an odd address is specified, the address is rounded up to the nearest even value. The default is zero.

ENDING ADDRESS:
The integer value which is the ending address of the memory area to be searched. The default address is the end of task.

11.3.6 Controlled Task Commands
The control commands allow control and trace execution of instructions in a task until:

- The execution of a specified number of instructions has been simulated.
- A specified address is placed in the PC.
- A breakpoint or simulated breakpoint occurs.

11.3.6.1 Assign Simulated Breakpoint — ASB. The ASB command is used to set up a breakpoint on a range of values for memory as follows:

- Memory alteration (A)
- CRU access (C)
- Program Counter value (P)
- Memory references (R)
- Status register value (S)

A memory write operation, which does not change the value in memory, is not a memory alteration. The breakpoints set with this command are only valid during a Simulate command. Breakpoints, in this case, are conditions which stop execution but allow execution to be resumed by an operator command, either by using the Resume Simulated Task (RST) command or by pressing the F3 function key. Each simulated breakpoint is assigned a number which is displayed at the completion of the ASB command. When a breakpoint occurs during simulation, a panel and the breakpoint number are displayed along with the display string.

Prompts:

```
ASSIGN SIMULATED BREAKPOINT
                 ON (A,C,P,R,S):  {A/C/P/R/S}        (PC)
                           FROM:  full exp
                           THRU:  [full exp]
                          COUNT:  full exp           (1)
                        DISPLAY:  [full exp]
```

Prompt Details:

ON (A,C,P,R,S):
The characters A, C, P, R, S are valid responses to this prompt and have the following meanings:

A = Memory alteration
C = CRU access
P = Program Counter (PC) value
R = Reference (memory)
S = Status Register (ST) value

FROM:
The integer expression that specifies the lower address limit for breakpointing.

THRU:
The integer expression that specifies the upper address limit for breakpointing. The default value is the value specified for the FROM prompt.

COUNT:
The integer expression that specifies the number of times this breakpoint is to be encountered before execution is halted. The default value is one.

DISPLAY:
The integer expression that specifies the memory address to be displayed when this breakpoint is reached. The default value is the PC value at the time the breakpoint is reached.

11.3.6.2 Delete Simulated Breakpoints — DSB. The DSB command is used to allow the user to delete a list of simulated breakpoints assigned with the Assign Simulated Breakpoint (ASB) command.

Prompts:

```
DELETE SIMULATED BREAKPOINTS
        BREAKPOINT NUMBERS:  [full exp list/ALL]
```

Prompt Details:

BREAKPOINT NUMBERS:

The integer value that specifies the number of the breakpoint to delete, which was the breakpoint number returned by the ASB command. If the characters ALL are entered, all the simulated breakpoints are deleted. The default is the breakpoint at which the task is stopped. Current simulated breakpoints can be obtained by executing the List Simulated Breakpoints (LSB) command.

11.3.6.3 List Simulated Breakpoints — LSB. The LSB command is used to display all current simulated breakpoints. When the breakpoints are listed, the first column of the display lists the numbers assigned when the breakpoints were set; the numbers start at one and are consecutive. The TYPE column lists letters for the ON prompt of an Assign Simulated Breakpoint command to identify the value on which the breakpoint was set, and the FROM and THRU columns list the corresponding operand addresses. The COUNT column lists the count operand entered when the breakpoints were set, and the REMAINING column lists the number of times the program has yet to go through the breakpoint. The DISPLAY column lists the display operand.

When the operands represent CRU addresses or ST register values, the operands are listed as hexadecimal numbers.

Prompts:

None

11.3.6.4 Quit Debug Mode — QD. The QD command is used to take a controlled task out of debug mode. The user has the option of killing the task at this point. If the user chooses not to kill the task, it will be left unconditionally suspended; but the user may still issue any of the general SCI commands. The Resume Task (RT) or Proceed from Breakpoint (PB) commands (depending on whether the task is at a breakpoint) may be used to activate the task.

The RT command is discussed in the task control commands paragraphs.

Prompts:

```
QUIT DEBUG MODE
                  KILL TASK ?:  YES/NO  (YES)
```

Prompt Details:

KILL TASK?:

If YES is entered, the currently executing task is killed. The task then executes its end-action routine. If NO is entered, the currently executing task is unconditionally suspended.

11.3.6.5 Resume Simulated Task — RST. The RST command is used to allow the user to resume simulation following a breakpoint, a simulated breakpoint, or simulation of a specified number of instructions. The last entered values for the FOR: and TO: prompts of the Simulate Task (ST) command are used as the RST limits. Upon reaching a terminating condition (breakpoint, simulated breakpoint, time-out, or the value specified for the TO: prompt), a panel and termination reason are displayed. Simulation can be continued by pressing the F3 function key or terminated by pressing the Command key, which returns SCI to the command mode. It is assumed that the task has been placed in controlled mode by the execution of the XD command, and that the task has reached a terminating condition.

Example:

In the following example, the RST command resumes a simulated task which has encountered a terminating condition, using the values entered for FOR and TO prompts of the ST command.

```
[ ] RST
RESUME SIMULATED TASK
```

Special Cases:

If the ST command has not been executed previously, the default value of FOR is one. One instruction is executed before the task is halted.

11.3.6.6 Simulate Task — ST. The ST command is used to provide controlled and traced execution of the instructions in a task. Controlled execution continues until the execution of a specified number of instructions has been simulated or until a specified address is placed in the PC or until a breakpoint or simulated breakpoint occurs. Simulation may be continued by pressing the F3 function key.

Simulated execution continues without operator intervention and locks out further SCI commands. Following simulation of the instruction whose address is specified by the response to the TO: prompt, SCI displays the panel and halts simulation. The user can regain SCI capabilities by pressing the Command key to return to command mode.

When the number of specified simulations has been performed, SCI displays the following message and halts simulation:

TIME OUT

Prompts:

SIMULATE TASK

```
                    FOR:  [full exp]  (*)
                   FROM:  [full exp]
                     TO:  [full exp]
```

Prompt Details:

FOR:
The integer expression that specifies the number of instruction simulations to be performed and must be less than or equal to 32,767. When the specified number of simulations has been performed, SCI displays the following message and halts simulation:

TIME OUT
If a null response is entered for this prompt, the value specified in a previous ST command is used; if no previous ST commands were executed, a one is used.

FROM:
The integer expression that specifies the address of the first instruction to be simulated. If a null response is entered in response to this prompt, simulation begins at the instruction with an address in the PC.

TO:
The integer expression that specifies the address of the last instruction to be simulated. The integer expression entered may be less than that entered for the FROM command. If a null response is entered in response to this prompt, simulation continues until a breakpoint or simulated breakpoint is encountered or until the user presses the Command key, returning SCI to command mode.

Messages:

STOP AT TRAP NO. X

where X is the number of the simulated breakpoint set through the Assign Simulated Breakpoint (ASB) command.

11.4 STATION DEPENDENT DISPLAYS

As mentioned previously, the displays generated by debugging SCI commands vary in format and content depending on the display device. High-speed display terminals (such as video display terminals) display more information than slower, hard-copy terminals. Table 11-2 lists the display generated by several of the debug commands in varying environments.

Table 11-2. Command Displays

Command	Hard Copy Regular	Hard Copy Debug	VDT Regular	VDT Debug
AB	—	—	—	PANEL
DB	—	—	—	PANEL
PB	—	—	—	PANEL
DBP	—	—	—	PANEL
LB	BRKPTS	BRKPTS	BRKPTS	BRKPTS
HT	—	—	—	PANEL
RT	—	—	—	PANEL
MM	INTERACT	INTERACT	INTERACT	INTERACT PANEL
LM	TLF	TLF	TLF	TLF
FW	MSG OR TLF	MSG OR TLF	MSG OR TLF	MSG + PANEL OR TLF
FB	MSG OR TLF	MSG OR TLF	MSG OR TLF	MSG + PANEL OR TLF
SIR	INT REG	INT REG	PANEL	PANEL
MWR	INTERACT	INTERACT	INTERACT	INTERACT PANEL
SWR	WKSPC	WKSPC	PANEL	PANEL
SP	PANEL	PANEL	PANEL	PANEL
SV	VALUES	VALUES	VALUES	VALUES
XD	—	—	—	PANEL
ASB	—	—	—	BRKPT NO. + PANEL
DSB	—	—	—	PANEL
LSB	—	SIMULATED BRKPTS	—	SIMULATED BRKPTS
ST	—	TRAP#OR'TIMEOUT'	—	TRAP#OR'TIMEOUT' + PANEL
RST	—	TRAP#OR'TIMEOUT'	—	TRAP#OR'TIMEOUT' + PANEL
QD	—	—	—	—

BRKPTS = Breakpoints
INTERACT = Interactive
INT = Internal registers
MSG = Message
PANEL = Debug panel
TIMEOUT = Time-out
TLF = Terminal local file
TRAP#OR = Trap number
WKSPC = Workspace

12

International Considerations of DX10

12.1 INTRODUCTION

DX10 is an international operating system designed to meet the commercial requirements of the United States as well as most Western European countries and Japan.

The international capabilities include the following:

- Data input and output on international peripheral devices designed to meet the language requirements of the following countries:

  Arabic-speaking countries
  Denmark/Norway
  France/Belgium
  Germany/Austria
  Japan
  Spanish-speaking countries
  Sweden/Finland
  Switzerland
  United Kingdom
  United States

- The ability to store and manipulate international data on any of the standard DX10 file types.

12.2 COUNTRY CODE

DX10 interprets data based on the country code selected during system generation. This code reflects the nationality of the data terminals attached to the system and defines system handling of the following areas:

- Handling of user data
- Handling of I/O data by the device service routines (DSRs)
- Handling of user keys by the KIF file management system.

Specify the country code during system generation by typing in the name of the country, or a unique abbreviation. The abbreviation need only be unique among the possible countries supported. The following table gives examples of unique country codes that satisfy the requirements for the system generation COUNTRY CODE prompt:

Country	Response to Country Code Prompt	Hexadecimal Country Code
USA - United States	US	0000
UK - United Kingdom	UK	0100
England - UK	EN	0100
Japan - Nippon	JA	0200
Belgium/France	BE	0300
France Data Processing/Belgium	F	0300
Denmark/Norway	DE	0400
Norway/Denmark	NOR	0400
Arabic-speaking countries	AR	0500
Reserved		0600
Reserved		0700
Reserved		0800
Austria/Germany	AU	0900
Germany/Austria	GE	0900
Finland/Sweden	FI	0A00
Sweden/Finland	SW	0A00
Spanish Speaking Countries	SP	0B00
FWP - French Word Processing	FWP	0C00
Switzerland	SWI	0D00

NOTE

Only use French Word Processing if you have the French Word Processing keyboard. Otherwise use France Data Processing as a reply to the COUNTRY CODE prompt during system generation. For countries not listed, use the default value (US).

User programs can determine the country code assigned to their system by executing a Retrieve System Information (>3F) SVC described in Section 8. (You can also use the SCC SCI command.)

12.3 INFORMATION INTERCHANGE CODES

DX10 and its supported peripheral devices use the internationally accepted information interchange codes for the various countries supported. In most cases, this is a seven-bit ASCII-type code with a few differences, generally, at the end of each table to handle the particularities of the local language.

Japan uses an eight-bit code to represent both the Latin alphabet (A-Z) and the Japanese character set in one combined information interchange code (JISCII).

The Japanese Model 911 VDT, which supports this extended character set, does not have the high/low intensity features available on the other versions of the Model 911.

The international interchange codes (KIF collating sequences) supported under DX10 are listed in Appendix B.

12.4 OPTIONAL SCI PROMPT

Since the latter part of the international ASCII-type tables is often used to handle local character extensions, certain of the special characters available in the USASCII code, such as the [ ] symbols, are sacrificed to these extension characters. For example, users who have a German Model 911 VDT will have the characters ÄÜ as the default SCI prompt instead of the more commonly displayed [ ] symbols. However, this problem can be overcome by the .OPTION SCI primitive which permits users to specify their own prompt characters in place of the default values (see the *DX10 Operations Guide* (Volume II)).

12.5 KEY INDEXED FILE COLLATING SEQUENCES

DX10's key indexed file (KIF) manager sorts user keys alphabetically (rather than according to the hexadecimal value of the letter), in creating the user's key indexed file. For German, Swedish, and Finnish, Spanish, Swiss, and France word processing, the alphabetic order of the language is different from English and a special collating sequence must be applied.

The collating sequence applied by the key indexed files is determined by the Country Code set for the system. For example, the key letters V, Ü, U, and D will be sorted by KIF as D, U, Ü, and V in a German System.

Key Order on Disk	Hexadecimal Value
44	D
U	55
Ü	5D
V	56

This implies that once a key indexed file is created on a German system, the collating sequence in which the data is arranged is only valid for a German system. For example, if the file were to be physically transferred to a Swedish system and a Show File command executed, the data would appear reasonably legible, although the Ü would appear as an Å on the Swedish VDT. The alphabetic sorting order of the keys will, however, be incorrect for the Swedish system (the Swedish Å should not appear between the U and the V in the collated access list). If an attempt is made to add data to the file, the key collating sequence will be disrupted and the file rendered unusable.

To ensure that this does not happen, users who wish to transfer a key indexed file between different international systems must first convert the file to a sequential file (using the Copy Key Indexed File to Sequential File command) then recreate the key indexed file on the new system (using the Copy Sequential File to Key Indexed File command) The new key indexed file will then be recreated following the collating rules of the new system.

The collating sequences for the various countries supported by DX10 are provided in Appendix B.

12.6 INTERNATIONAL DEVICES

The following devices are available in international form for United Kingdom, Germany, Belgium, France, Denmark, Norway, Sweden, Finland, and Japan. Contact your service representative for other international devices.

- Model 743 Data Terminal
- Model 745 Data Terminal
- Model 763 Data Terminal (not Japan)
- Model 765 Data Terminal (not Japan)
- Model 781 RO Data Terminal
- Model 783 KSR Data Terminal
- Model 785 Communications Terminal
- Model 787 Communications Terminal
- Model 820 RO/KSR Data Terminal

Table 12-1 shows characters added and characters removed to internationalize teleprinter devices.

Table 12-1. DSR Codes for International Characters

ASCII CODE

COUNTRY	23	40	5B	5C	5D	5E	60	7B	7C	7D	7E
US ASCII STANDARD	#	@	[	\	]	∧	`	{	¦	}	~
UNITED KINGDOM	£	@	[	\	]	∧	`	{	¦	}	~
GERMANY/AUSTRIA	#	@	Ä	Ö	Ü	∧	`	ä	ö	ü	β
SWEDEN/FINLAND *	#	É	Ä	Ö	Å	Ü	é	ä	ö	å	ü
NORWAY/DENMARK	#	@	Æ	Ø	Å	∧	`	æ	ø	å	~
SPANISH-SPEAKING	#	@	¡	Ñ	¿	∧	`	°	ñ	ç	~
SWITZERLAND	£	à	é	ç	è	∧	`	ä	ö	ü	¨
FRANCE **											
FRENCH WP	£	à	°	ç	§	∧	`	é	ù	è	¨
ITALY **											
HOLLAND **											

*¤ USES ASCII CODE >24, REPLACING THE $ CHARACTER IN US ASCII.

** USES THE US ASCII STANDARD.

JISCII CODE	5C	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC
JAPANESE CHARACTER	¥	｡	｢	｣	､	･	ｦ	ｧ	ｨ	ｩ	ｪ	ｫ	ｬ

JISCII CODE	AD	AE	AF	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9
JAPANESE CHARACTER	ｭ	ｮ	ｯ	ｰ	ｱ	ｲ	ｳ	ｴ	ｵ	ｶ	ｷ	ｸ	ｹ

JISCII CODE	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3	C4	C5	C6
JAPANESE CHARACTER	ｺ	ｻ	ｼ	ｽ	ｾ	ｿ	ﾀ	ﾁ	ﾂ	ﾃ	ﾄ	ﾅ	ﾆ

JISCII CODE	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3
JAPANESE CHARACTER	ﾇ	ﾈ	ﾉ	ﾊ	ﾋ	ﾌ	ﾍ	ﾎ	ﾏ	ﾐ	ﾑ	ﾒ	ﾓ

JISCII CODE	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	
JAPANESE CHARACTER	ﾔ	ﾕ	ﾖ	ﾗ	ﾘ	ﾙ	ﾚ	ﾛ	ﾜ	ﾝ	ﾞ	ﾟ	

12.7 IPF — INTERNATIONAL PRINT FILE COMMAND

The International Print File (IPF) command formats output files to match the requirements of the printwheel or printer type to be used and places the file in the output queue for the selected printer. This requires the printer to be internationalized to support the country's character set.

Default conversion tables can be selected to convert DX10 files for output to an LQ45 printer with either LQ45 FRANCE or LQ45 DEUTSCHLAND printwheels. No conversion is required for the other countries supported by DX10, provided the standard 911 international keyboard for that country is used with the appropriate national printwheel. Users can create conversion files for other printwheels, printers, languages, or overstrike requirements.

Users can create output formatting files to match raw data output to the code requirements of any type of printer or to specify overstrike characters. These output formatting files must be created using the following data format:

Record	File Column 1	Contents
1	XX	Country code (two characters)
2	YY	Buffer coefficient × 10 (two characters)
3	Z1, Z2,. . ., Zn	List of characters affected by backspace. None if left blank.
4	VALUE 0	Code to be substituted for ASCII >00
5	VALUE 1	Code to be substituted for ASCII >01
.		
.		
.		
131		

12.7.1 IPF Command Format

The following example shows the IPF command prompts and response types:

```
IPF
INTERNATIONAL PRINT FILE
              INPUT FILE PATHNAME:  pathname@
             OUTPUT FILE PATHNAME:  pathname@
                          COUNTRY:  pathname@
                     ANSI FORMAT?:  [yes/no]
                  LISTING DEVICE:  pathname@
  DELETE INPUT AFTER PRINTING?:  [yes/no]
 DELETE OUTPUT AFTER PRINTING?:  [yes/no]
            NUMBER OF LINES/PAGE:  [int]
```

12.7.2 IPF Command User Responses

The prompt types and valid responses are as follows:

Field Prompts	Response Required or Optional	User Responses
INPUT FILE PATHNAME:	R	A pathname that identifies the the file to be printed. For TIPE users, the command is used to format the user's file then place the file in an output file. The pathname of this file is specified as the INPUT FILE PATHNAME to the IPF command.
OUTPUT FILE PATHNAME:	R	A device name of a device or the pathname of a file to which the information should be printed.
COUNTRY:	R	A pathname of a formatting table used to process the output file into the required printing format. If F is entered, the formatting table for the LQ45 FRANCE printwheel is used. This causes the circumflex (Λ) and the umlaut (¨) to be typed above any character they precede in the output file. If D is entered, the formatting table for the LQ45 DEUTSCHLAND printwheel is used. This matches the German character set used in DX10 to that of the LQ45 (see Appendix B for character sets).
ANSI FORMAT?:	R	Y (yes) indicates that the file being printed contains ANSI carriage control characters. NO specifies that the file does not contain ANSI carriage control characters. The ANSI control character (the first portion of each record) has the following meanings: blank — single space before printing 0 — double space before printing 1 — top of page before printing + — suppress line space (overprint) The initial value is YES. TIPE users should accept the intial value.

Field Prompts	Response Required or Optional	User Responses
LISTING DEVICE:	R	The device name of the device to which the contents of the file(s) is printed.
DELETE INPUT AFTER PRINTING?	O	Y (yes) indicates that the original output file is to be deleted after it is formatted. NO specifies that the original output file should not be deleted. The initial value is NO.
DELETE OUTPUT AFTER PRINTING?:	O	Y (yes) indicates the file being printed is to be deleted after it is printed. NO specifies that the file should not be deleted. The initial value is YES.
NUMBER OF LINES/PAGE:	O	The number of lines to be printed on one page. The default value is 62.

12.7.3 IPF Command Example

In the following example, the contents of the file VOL1.DIR2.FRENCH1 is printed on the LQ45 letter quality printer with 55 lines per page.

```
[ ]IPF
INTERNATIONAL PRINT FILE
                    FILE PATHNAME:  VOL1.DIR2.FRENCH1
             OUTPUT FILE PATHNAME:  .TIPE03
                          COUNTRY:  F
                     ANSI FORMAT?:  YES
                   LISTING DEVICE:  ST04
            DELETE AFTER PRINTING?: NO
     DELETE OUTPUT AFTER PRINTING?: YES
             NUMBER OF LINES/PAGE:  55
```

Test d'un texte français utilisant les mots cléfs suivants:

Noël, élève, tête, français, gîte, hôtel, à, où, était, pâte, maïs, flûte.

La tabutation utilisée est de 20 à 50.

A Noël les élèves en tête de classe de français eurent un gîte fourni dans un hôtel à Nice, où le menu principal était à base de pâtes et de maïs avec heureusement une flûte de Champagne.

Ce texte doit continuer sur la page suivante.

12.7.4 IPF Error Messages

The following error messages are unique to the International Print File command.

Error Code	Text	Reason
01	IPF ERROR	Buffer coefficient not numeric.
02	IPF ERROR	Buffer coefficient less than 10.
03	COUNTRY FILE ERROR	Conversion table does not contain either 128 or 256 entries.
04	INPUT FILE ERROR	Input file is not sequential.
05	IPF ERROR	Unknown character found in input file.

12.8 SCC — SHOW COUNTRY CODE COMMAND

The SCC command lists the country code selected during system generation. (Refer to paragraph 12.2 for a list of valid country codes.) The response to the LISTING ACCESS NAME prompt must be a legal device or file name.

12.8.1 Command Format

The format of the SCC command is as follows:

```
SHOW COUNTRY CODE
          LISTING ACCESS NAME:  pathname@
```

12.8.2 SCC Command User Responses

System Prompts	Response Required or Optional	User Responses
LISTING ACCESS NAME:	R	The device or file name to receive the list of country codes. If a null response is entered, the list is displayed on the terminal screen.

12.8.3 SCC Command Example

In the following example, the SCC command lists the country code US (United States) to the terminal local file.

```
[ ]SCC
SHOW COUNTRY CODE
          LISTING ACCESS NAME:  LP01
```

Related Commands.

MCC (Modify Country Code)

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[1] 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	(blank key)	SHIFT, LINE INS CHAR	(blank key)	(blank key)	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)

Table A-2. Frequently Used Key Sequences

Function	Key Sequence
Log-on	Attention/(Shift)!
Hard-break	Attention/(Control)x
Hold	Attention
Resume	Any key

Table A-3. 911 Keycap Name Equivalents

911 Phrase	Generic Name
Blank gray	Initialize Input
Blank orange	Attention
Down arrow	Next Line
Escape	Exit
Left arrow	Previous Character
Right arrow	Next Character
Up arrow	Previous Line

2284734 (8/14)

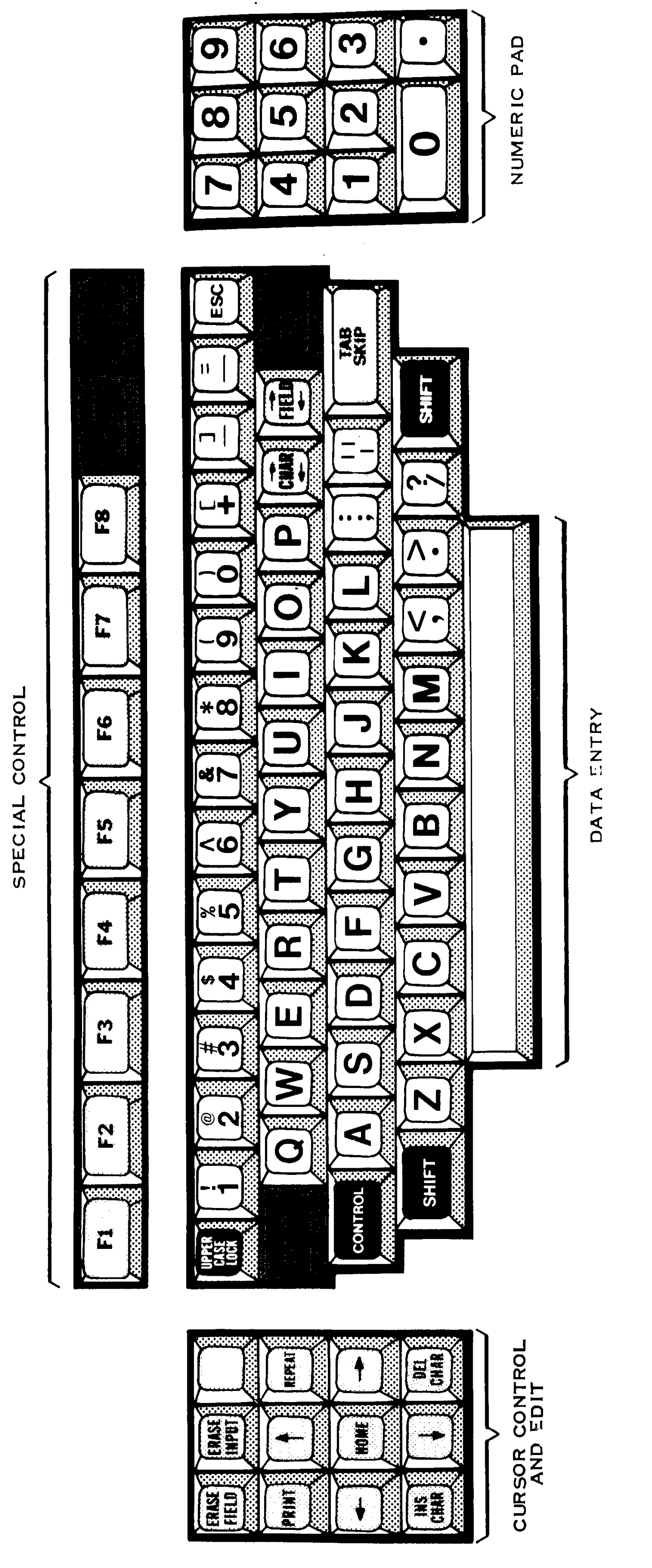


Figure A-1. 911 VDT Standard Keyboard Layout

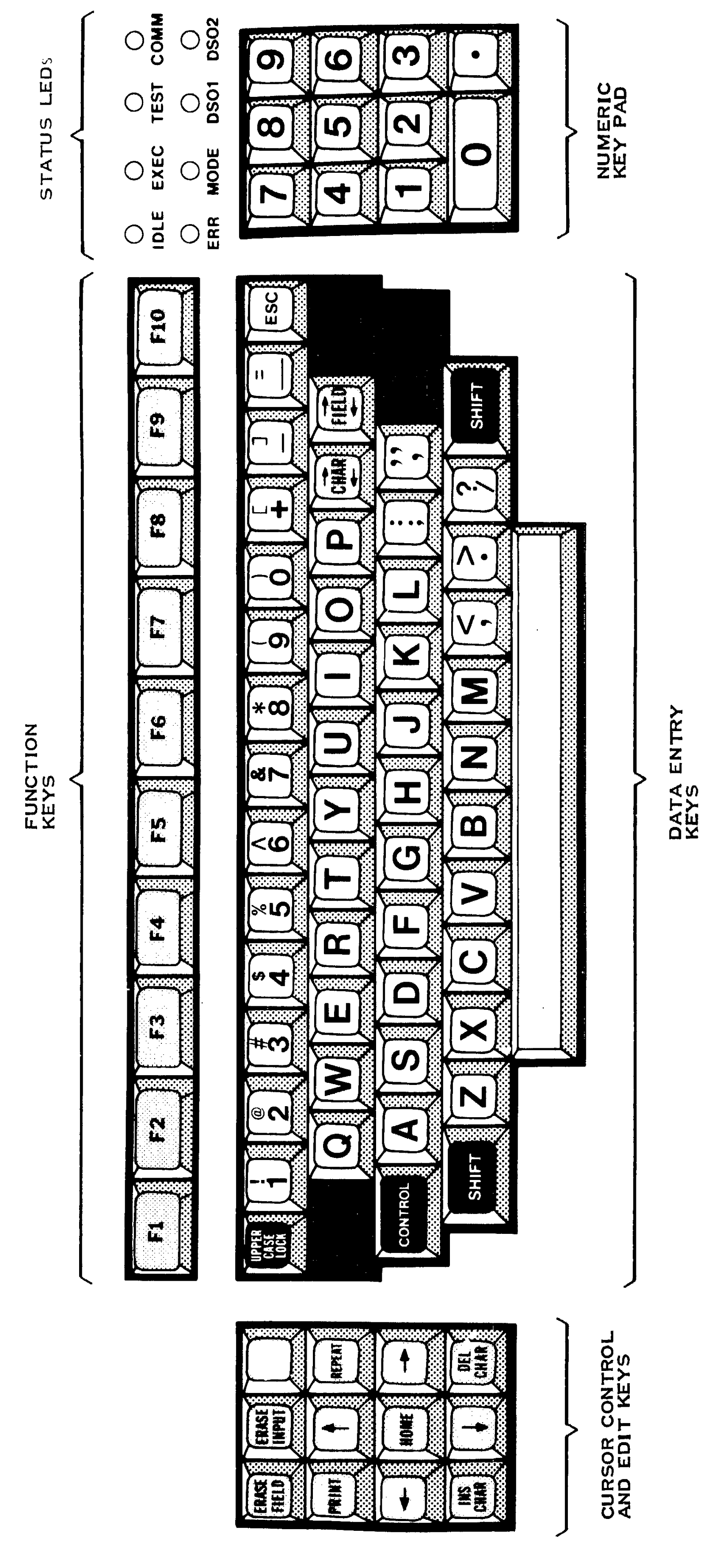


Figure A-2. 915 VDT Standard Keyboard Layout

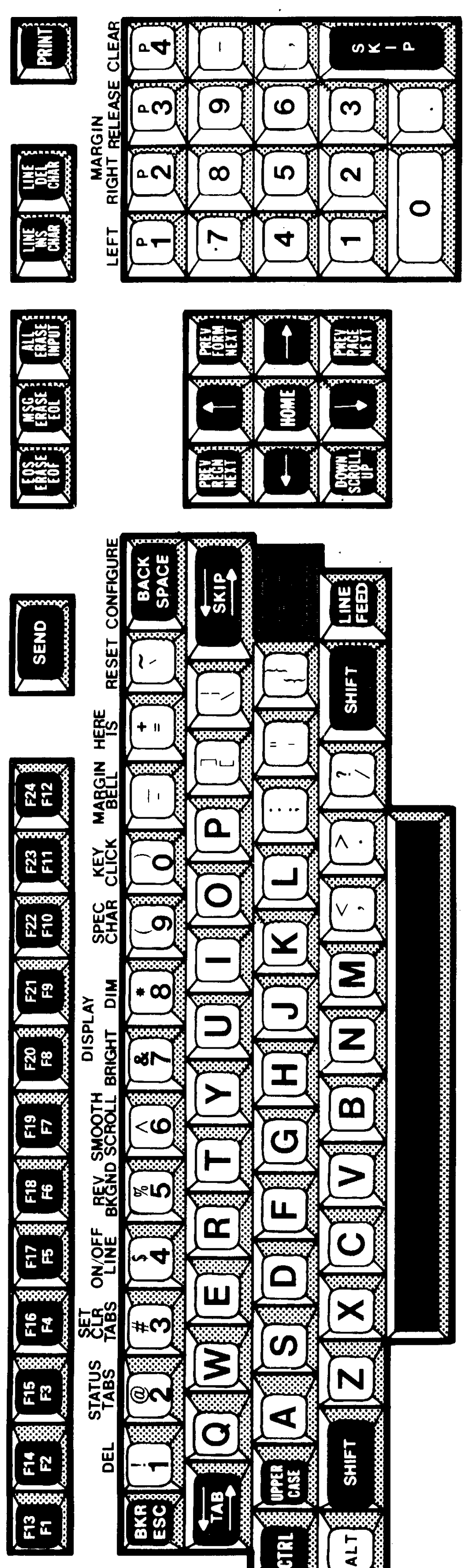


Figure A-3. 940 EVT Standard Keyboard Layout

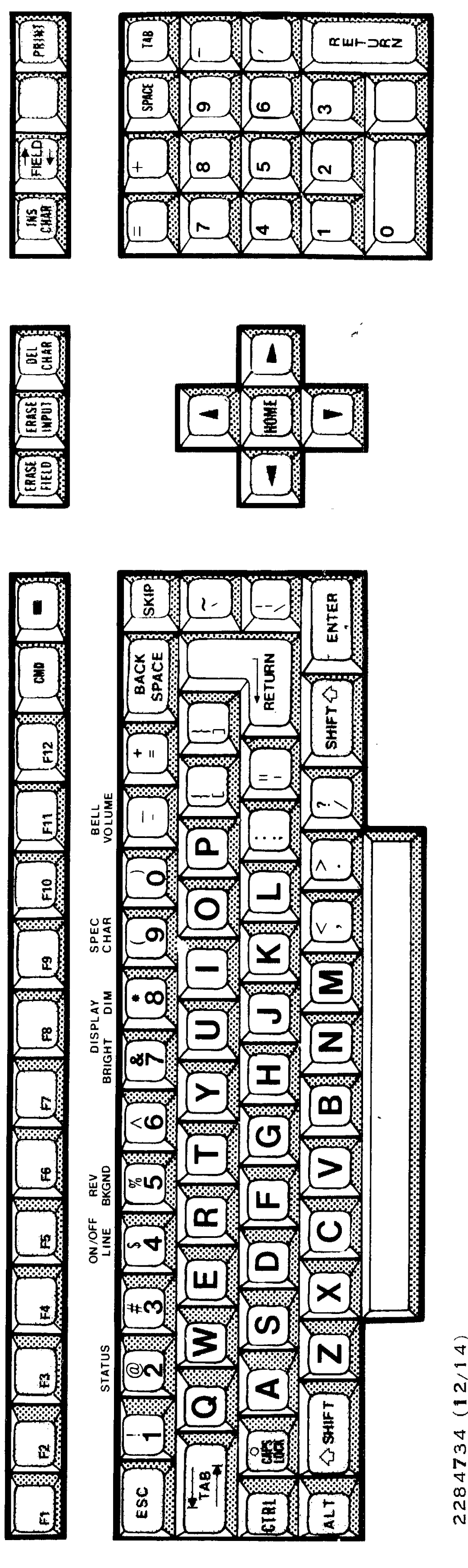


Figure A-4. 931 VDT Standard Keyboard Layout

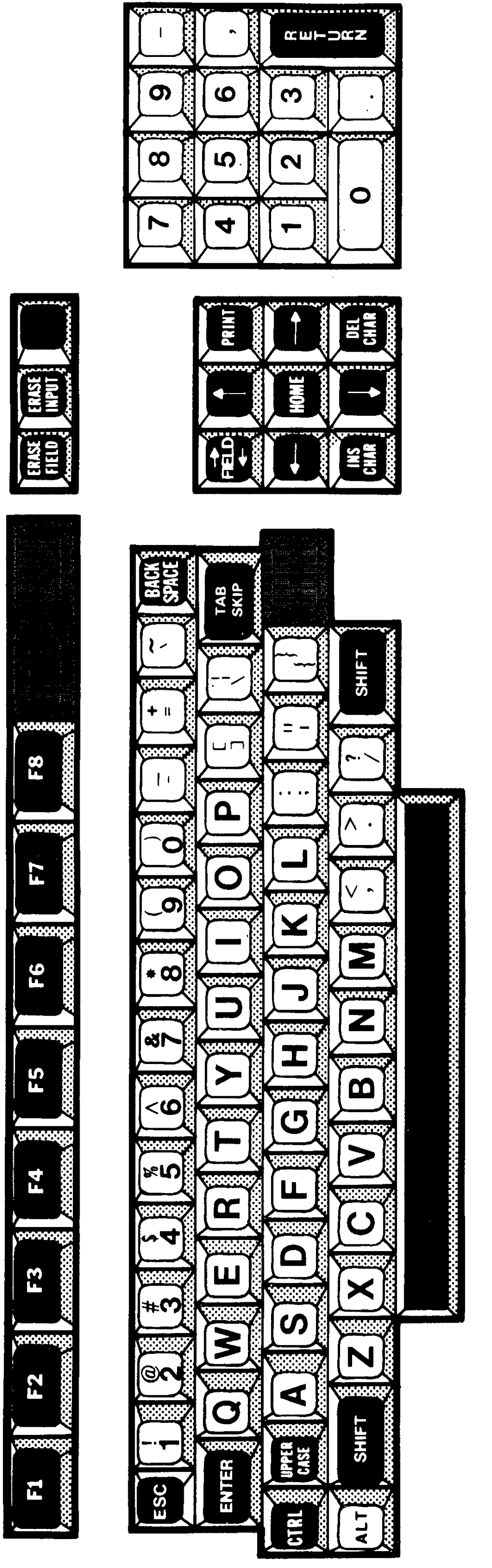


Figure A-5. Business System Terminal Standard Keyboard Layout

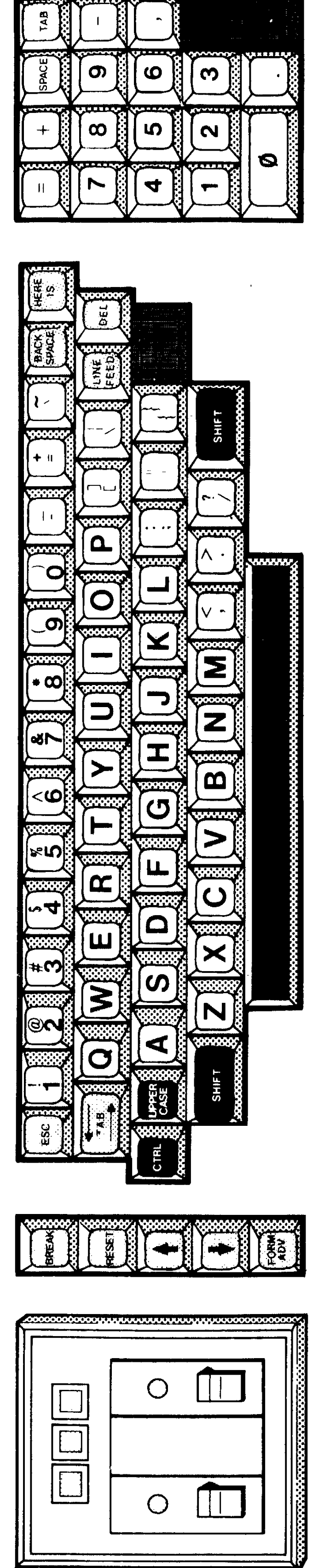


Figure A-6. 820 KSR Standard Keyboard Layout

2284734 (14/14)

Appendix B

ASCII Device I/O Operations

B.1 GENERAL

DX10 supports ASCII I/O operations with all devices. This appendix consists of tables that show key codes generated by supported terminals and what the DSRs do with the codes according to the requested operation. It also includes tables that show graphics character sets and a table that shows KIF collating sequences according to country code.

B.2 CHARACTER TYPES

Tables B-1 through B-6 show how the DSR treats the ASCII characters for each key on the 911 VDT, 931 VDT, 940 EVT, Business System terminal, 820 KSR, and 783 TPD, respectively. The 783 TPD is representative of the TPD class of terminals. The column headings for each table mean the following:

ASCII Character
: The defined ASCII character. Characters >00 through >1F and >7F are control characters and are not printable.

Terminal Key
: The key you press on the terminal to generate the ASCII character.

Terminal Generated Code
: The hexadecimal code generated by the terminal for that key. In the notation A/B, A is the code when the terminal is set to transmit even parity, and B is the code when the terminal is set to transmit odd parity. For the TPD DSR, this code is returned to your task when the device is in eight-bit ASCII mode and your task issues a Read Direct operation with the eight-bit data option. In pass thru mode on the 931, 940, and Business System terminals, this code is returned to your task, except for DC1 and DC3, which the DSR never returns. The 931, 940, and Business System terminals generate two- or three-character escape sequences for some keys. For example, the F1 key on the 931 generates 1B6931 (ESC i 1).

Type
: Indicates one of the following:

S = System edit character
E = Event key
T = Task edit character
H = Hold character
F = End-of-file
R = End-of-record

ASCII Task Edit

Lists the codes returned to your task for this key when your task is doing a Read ASCII operation with the task edit flag set and the device has been opened in event key mode.

Buffer

The code placed in the read buffer. If there is no entry, no code is placed in the read buffer.

Event

The code placed in the event byte of the call block. If there is no entry, no code is placed in the event byte.

Flag

The flag or flags set in the system flags byte of the call block when this key is received. EVT means the event flag is set. EOF means the end-of-file flag is set.

ASCII

The code returned to your task for this key when your task is doing a simple Read ASCII operation with no special flags set. If there is no entry in this column, nothing is returned.

Direct

The code returned to your task for this key when your task is doing a Read Direct operation with no special flags set. This column is not applicable to a 911, 931, 940, or Business System terminal since a Read Direct on these terminals does not interact with the keyboard.

Figures B-1 through B-3 show graphics character sets. To read or write graphics characters, an extended call block must be used with the graphics bit set in the extended flags. Figure B-1 shows the keyboard positions of the keys that produce graphics on the 911.

Figure B-2 shows the graphics characters for the 931. The chart on the left shows the characters and the ASCII codes they produce. For example, an uppercase A produces ASCII code >41. The top row contains the first digit of the code; the left column contains the second digit. If you are using the special character set, the chart on the right shows the graphics characters and the ASCII codes they produce. For example, the question mark (?) key produces the graphics character with the ASCII code >3F.

Figure B-3 shows the graphics characters for the 940. The chart on the left shows the graphics characters and the ASCII codes they produce. For example, a left arrow (←) produces ASCII code >4E. The top row contains the first digit of the code; the left column contains the second digit. The chart on the left shows the keyboard positions of the keys that produce these graphics characters.

Table B-7 shows the graphics character sets for the display terminals.

Table B-1. 911 VDT Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
NULL	(CONTROL) 3	00		20	80		00	N/A*
SOH	(CONTROL) A	01		20	81		00	
STX	(CONTROL) B	02		20	82		00	
ETX	(CONTROL) C	03		20	83		00	
EOT	(CONTROL) D	04		20	84		00	
ENQ	(CONTROL) E	05		20	85		00	
ACK	(CONTROL) F	06		20	86		00	
BELL	(CONTROL) G	07		20	87			
BS	(CONTROL) H	08						
HT	(CONTROL) I	09		20	89			
LF	(CONTROL) J	0A		20	8A			
VT	(CONTROL) K	0B		20	8B			
FF	(CONTROL) L	0C		20	8C			
CR	(CONTROL) M	0D		20	8D		00	
CR	RETURN	0D	STR	20	8D		00	
SO	(CONTROL) N	0E		20	8E			
SI	(CONTROL) O	0F		20	8F			
DLE	(CONTROL) P	10						
DC1	(CONTROL) Q	11						
DC2	(CONTROL) R	12		20	20		00	
DC3	(CONTROL) S	13	F	20	93	EOF	00 EOF	
DC4	(CONTROL) T	14		20	94			
NAK	(CONTROL) U	15		20	95			
SYN	(CONTROL) V	16		20	96		00	
ETB	(CONTROL) W	17		20	97		00	
CAN	(CONTROL) X	18		20	98		00	
EM	(CONTROL) Y	19		20	99		00	
SUB	(CONTROL) Z	1A		20	9A		00	
ESC	ESC	1B		20	9B		00	
FS	(CONTROL) ,	1C		20	9C		00	
GS	(CONTROL) +	1D		20	9D		00	
RS	(CONTROL) .	1E		20	9E		00	
US	(CONTROL) /	1F		20	9F		00	
Space	Space bar	20		20	20		20	
!	!	21		21	21		21	
"	"	22		22	22		22	
#	#	23		23	23		23	
$	$	24		24	24		24	
%	%	25		25	25		25	
&	&	26		26	26		26	
'	'	27		27	27		27	
(	(	28		28	28		28	
)	)	29		29	29		29	
*	*	2A		2A	2A		2A	
+	+	2B		2B	2B		2B	

Note:

* Not applicable; does not interact with keyboard

Table B-1. 911 VDT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit			ASCII	Direct
				Buffer	Event	Flag		
,	,	2C		2C	2C		2C	
-	-	2D		2D	2D		2D	
.	.	2E		2E	2E		2E	
/	/	2F		2F	2F		2F	
0	0	30		30	30		30	
1	1	31		31	31		31	
2	2	32		32	32		32	
3	3	33		33	33		33	
4	4	34		34	34		34	
5	5	35		35	35		35	
6	6	36		36	36		36	
7	7	37		37	37		37	
8	8	38		38	38		38	
9	9	39		39	39		39	
:	:	3A		3A	3A		3A	
;	;	3B		3B	3B		3B	
<	<	3C		3C	3C		3C	
=	=	3D		3D	3D		3D	
>	>	3E		3E	3E		3E	
?	?	3F		3F	3F		3F	
@	@	40		40	40		40	
A	A	41		41	41		41	
B	B	42		42	42		42	
C	C	43		43	43		43	
D	D	44		44	44		44	
E	E	45		45	45		45	
F	F	46		46	46		46	
G	G	47		47	47		47	
H	H	48		48	48		48	
I	I	49		49	49		49	
J	J	4A		4A	4A		4A	
K	K	4B		4B	4B		4B	
L	L	4C		4C	4C		4C	
M	M	4D		4D	4D		4D	
N	N	4E		4E	4E		4E	
O	O	4F		4F	4F		4F	
P	P	50		50	50		50	
Q	Q	51		51	51		51	
R	R	52		52	52		52	
S	S	53		53	53		53	
T	T	54		54	54		54	
U	U	55		55	55		55	
V	V	56		56	56		56	
W	W	57		57	57		57	
X	X	58		58	58		58	
Y	Y	59		59	59		59	
Z	Z	5A		5A	5A		5A	
[	[	5B		5B	5B		5B	
\	(CONTROL) _	5C		5C	5C		5C	

Table B-1. 911 VDT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit			ASCII	Direct
				Buffer	Event	Flag		
]	]	5D		5D	5D		5D	
^	^	5E		5E	5E		5E	
_	_	5F		5F	5F		5F	
`	(CONTROL) 9	60		60	60		60	
a	a	61		61	61		61	
b	b	62		62	62		62	
c	c	63		63	63		63	
d	d	64		64	64		64	
e	e	65		65	65		65	
f	f	66		66	66		66	
g	g	67		67	67		67	
h	h	68		68	68		68	
i	i	69		69	69		69	
j	j	6A		6A	6A		6A	
k	k	6B		6B	6B		6B	
l	l	6C		6C	6C		6C	
m	m	6D		6D	6D		6D	
n	n	6E		6E	6E		6E	
o	o	6F		6F	6F		6F	
p	p	70		70	70		70	
q	q	71		71	71		71	
r	r	72		72	72		72	
s	s	73		73	73		73	
t	t	74		74	74		74	
u	u	75		75	75		75	
v	v	76		76	76		76	
w	w	77		77	77		77	
x	x	78		78	78		78	
y	y	79		79	79		79	
z	z	7A		7A	7A		7A	
{	(CONTROL) ;	7B		7B	7B		7B	
¦	(CONTROL) 8	7C		7C	7C		7C	
}	(CONTROL) '	7D		7D	7D		7D	
~	(CONTROL) 0	7E		7E	7E		7E	
DEL	(CONTROL) -	7F						
	ERASE FIELD	80	S					
	ERASE INPUT	81	ST	20	8E			
	HOME	82	ST	20	8C			
	TAB	83	ST	20	89			
	DEL CHAR	84	S					
	SKIP	85	ST	20	8B			
	INS CHAR	86	S					
	FIELD left	87	T	20	94			
	Left arrow	88	S					
	CHAR left	88	S					
	Up arrow	89	T	20	95			
	Right arrow	8A	S	20	20		20	
	CHAR right	8A	S	20	20		20	
	Down arrow	8B	T	20	8A			

Table B-1. 911 VDT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit			ASCII	Direct
				Buffer	Event	Flag		
	FIELD right	8C	T	20	87			
	(CONTROL) 5	8D	E	20	9D	EVT		
	(CONTROL) 6	8E	E	20	9E	EVT		
	(CONTROL) 7	8F	E	20	9F	EVT		
	(CONTROL) 1	90	E	20	80	EVT		
	(CONTROL) 2	91	E	20	9A	EVT		
	F1	92	E	20	81	EVT		
	F2	93	E	20	82	EVT		
	F3	94	E	20	83	EVT		
	F4	95	E	20	84	EVT		
	F5	96	E	20	85	EVT		
	F6	97	E	20	86	EVT		
	F7	98	E	20	96	EVT		
	F8	99	E	20	97	EVT		
	PRINT	9A	E	20	99	EVT		
	CMD	9B	E	20	98	EVT		
	Blank orange	9C	H					
	Blank gray	9F	T	20	8F			
	ENTER	A0	TF	00 EOF	93	EOF		
	(CONTROL) 4	A1	EF	20	9C	EVT		

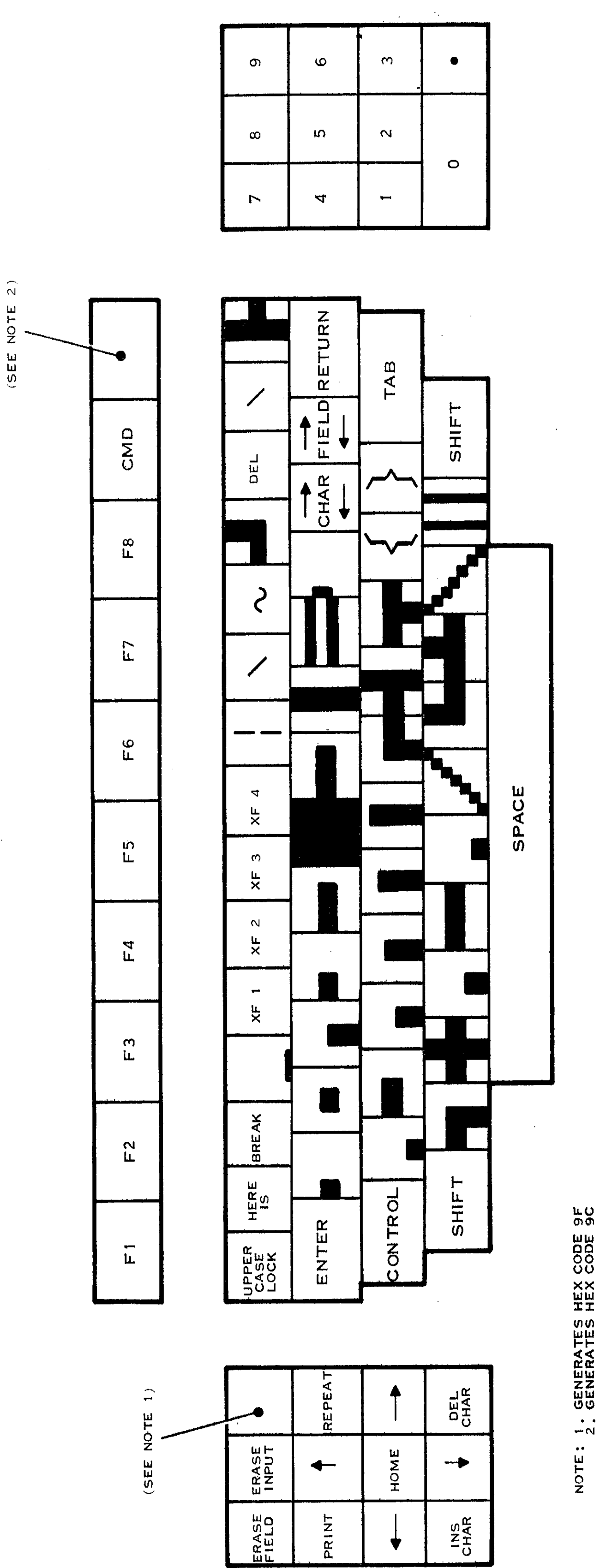


Figure B-1. 911 VDT Graphics Character Keyboard Positions

Table B-2. 931 VDT Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	ASCII Task Edit					
			Type	Buffer	Event	Flag	ASCII	Direct
NULL	(ALT) 1	00						
SOH	(CTRL) A	01						N/A[1]
STX	(CTRL) B	02						
ETX	(CTRL) C	03						
EOT	(CTRL) D	04						
ENQ	(CTRL) E	05						
ACK	(CTRL) F	06						
BELL	(CTRL) G	07						
BS	BACKSPACE	08						
BS	(CTRL) H	08						
HT	TAB	09		20	89			
HT	(CTRL) I	09		20	89		00	
LF	(CTRL) J	0A		20	87		00	
VT	(CTRL) K	0B						
FF	(CTRL) L	0C						
CR	(CTRL) M	0D		20	8D		00	
CR	RETURN	0D		20	8D		00	
CR	ENTER	0D	F	20	93	EOF	00 EOF	
SO	(CTRL) N	0E						
SI	(CTRL) O	0F						
DLE	(CTRL) P	10						
DC1	(CTRL) Q	11[2]						
DC2	(CTRL) R	12						
DC3	(CTRL) S	13[2]	H					
DC4	(CTRL) T	14						
NAK	(CTRL) U	15						
SYN	(CTRL) V	16						
ETB	(CTRL) W	17						
CAN	(CTRL) X	18		20	98		00	
EM	(CTRL) Y	19						
SUB	(CTRL) Z	1A						
ESC	ESC	1B						
ESC	(CTRL) [	1B						
FS	(CTRL) ,	1C						
GS	(CTRL)]	1D						
RS	(CTRL) .	E						
US	(CTRL) /	1F						
Space	Space bar	20		20	20		20	
!	!	21		21	21		21	
"	"	22		22	22		22	
#	#	23		23	23		23	
$	$	24		24	24		24	
%	%	25		25	25		25	
&	&	26		26	26		26	

Notes:

[1] Not applicable; does not interact with keyboard.

[2] The DSR recognizes DC1 and DC3 but it never returns them to the application.

Table B-2. 931 VDT Key Character Code Transformations (Continued)

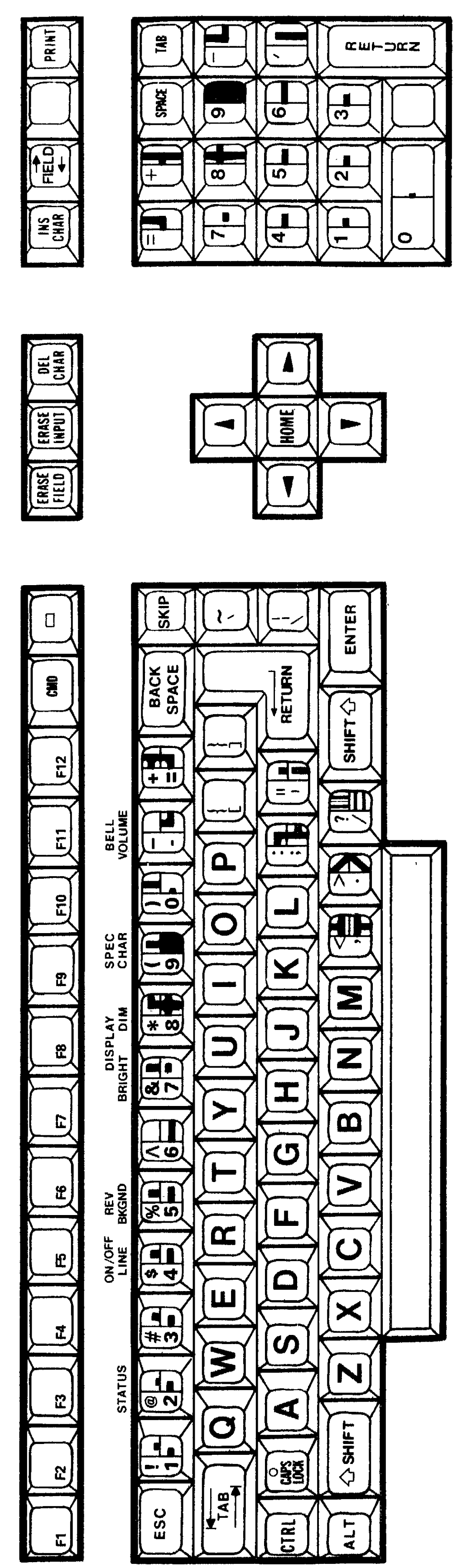
ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
'	'	27		27	27		27	
(	(	28		28	28		28	
)	)	29		29	29		29	
*	*	2A		2A	2A		2A	
+	+	2B		2B	2B		2B	
,	,	2C		2C	2C		2C	
-	-	2D		2D	2D		2D	
.	.	2E		2E	2E		2E	
/	/	2F		2F	2F		2F	
0	0	30		30	30		30	
1	1	31		31	31		31	
2	2	32		32	32		32	
3	3	33		33	33		33	
4	4	34		34	34		34	
5	5	35		35	35		35	
6	6	36		36	36		36	
7	7	37		37	37		37	
8	8	38		38	38		38	
9	9	39		39	39		39	
:	:	3A		3A	3A		3A	
;	;	3B		3B	3B		3B	
<	<	3C		3C	3C		3C	
=	=	3D		3D	3D		3D	
>	>	3E		3E	3E		3E	
?	?	3F		3F	3F		3F	
@	@	40		40	40		40	
A	A	41		41	41		41	
B	B	42		42	42		42	
C	C	43		43	43		43	
D	D	44		44	44		44	
E	E	45		45	45		45	
F	F	46		46	46		46	
G	G	47		47	47		47	
H	H	48		48	48		48	
I	I	49		49	49		49	
J	J	4A		4A	4A		4A	
K	K	4B		4B	4B		4B	
L	L	4C		4C	4C		4C	
M	M	4D		4D	4D		4D	
N	N	4E		4E	4E		4E	
O	O	4F		4F	4F		4F	
P	P	50		50	50		50	
Q	Q	51		51	51		51	
R	R	52		52	52		52	
S	S	53		53	53		53	
T	T	54		54	54		54	
U	U	55		55	55		55	
V	V	56		56	56		56	
W	W	57		57	57		57	

Table B-2. 931 VDT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
X	X	58		58	58		58	
Y	Y	59		59	59		59	
Z	Z	5A		5A	5A		5A	
[	[	5B		5B	5B		5B	
\	\	5C		5C	5C		5C	
]	]	5D		5D	5D		5D	
^	^	5E		5E	5E		5E	
_	_	5F		5F	5F		5F	
`	`	60		60	60		60	
a	a	61		61	61		61	
b	b	62		62	62		62	
c	c	63		63	63		63	
d	d	64		64	64		64	
e	e	65		65	65		65	
f	f	66		66	66		66	
g	g	67		67	67		67	
h	h	68		68	68		68	
i	i	69		69	69		69	
j	j	6A		6A	6A		6A	
k	k	6B		6B	6B		6B	
l	l	6C		6C	6C		6C	
m	m	6D		6D	6D		6D	
n	n	6E		6E	6E		6E	
o	o	6F		6F	6F		6F	
p	p	70		70	70		70	
q	q	71		71	71		71	
r	r	72		72	72		72	
s	s	73		73	73		73	
t	t	74		74	74		74	
u	u	75		75	75		75	
v	v	76		76	76		76	
w	w	77		77	77		77	
x	x	78		78	78		78	
y	y	79		79	79		79	
z	z	7A		7A	7A		7A	
{	{	7B		7B	7B		7B	
¦	¦	7C		7C	7C		7C	
}	}	7D		7D	7D		7D	
~	~	7E		7E	7E		7E	
	(SHIFT) TAB	1B32						
	(SHIFT) Blank gray	1B3C						
	ERASE FIELD	1B3D						
	Up arrow	1B41		20	95			
	Down arrow	1B42		20	8A			
	Right arrow	1B43		20	20		20	
	Left arrow	1B44						
	HOME	1B48		20	8C			
	(SHIFT) ERASE FIELD	1B49						
	(SHIFT) ERASE INPUT	1B4A						

Table B-2. 931 VDT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
	ERASE INPUT	1B4B		20	8E			
	(SHIFT) CMD	1B4C						
	Blank gray	1B4E		20	8F			
	(SHIFT) DEL CHAR	1B4F	S					
	INS CHAR	1B50	S					
	DEL CHAR	1B51	S					
	PRINT	1B57	STE	20	99	EVT		
	(SHIFT) Blank orange	1B65	S					
	(SHIFT) ESC	1B66	STE	20	9B		00	
	Blank orange	1B67	S					
	CMD	1B68	SE	20	98	EVT		
	F1	1B6931	E	20	81	EVT		
	F2	1B6932	E	20	82	EVT		
	F3	1B6933	E	20	83	EVT		
	F4	1B6934	E	20	84	EVT		
	F5	1B6935	E	20	85	EVT		
	F6	1B6936	E	20	86	EVT		
	F7	1B6937	E	20	96	EVT		
	F8	1B6938	E	20	97	EVT		
	F9	1B6939	E	20	80	EVT		
	F10	1B693A	E	20	9A	EVT		
	F11	1B693B	E	20	9C	EVT		
	F12	1B693C	E	20	9D	EVT		
	(SHIFT) F1	1B693D	E	20	9E	EVT		
	(SHIFT) F2	1B693E	E	20	9F	EVT		
	(SHIFT) F3	1B693F						
	(SHIFT) F4	1B6940						
	(SHIFT) F5	1B6941						
	(SHIFT) F6	1B6942						
	(SHIFT) F7	1B6943						
	(SHIFT) F8	1B6944						
	(SHIFT) F9	1B6945						
	(SHIFT) F10	1B6946						
	(SHIFT) F11	1B6947						
	(SHIFT) F12	1B6948						
	FIELD right	1B696F	ST	20	87			
	SKIP	1B73	ST	20	8B			
	FIELD left	1B74	ST	20	94			



2285394

Figure B-2. 931 VDT Graphics Characters

Table B-3. 940 EVT Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
NULL	(CTRL)`	00						N/A[1]
SOH	(CTRL) A	01						
STX	(CTRL) B	02						
ETX	(CTRL) C	03						
EOT	(CTRL) D	04						
ENQ	(CTRL) E	05						
ACK	(CTRL) F	06						
BELL	(CTRL) G	07						
BS	BACK SPACE	08						
BS	(CTRL) H	08						
HT	TAB right	09		20	89		00	
HT	(CTRL) I	09		20	89		00	
LF	LINE FEED	0A		20	87			
LF	(CTRL) J	0A		20	87			
VT	(CTRL) K	0B						
FF	(CTRL) L	0C						
CR	(CTRL) M	0D		20	8D		00	
CR	RETURN	0D		20	8D		00	
SO	(CTRL) N	0E						
SI	(CTRL) O	0F						
DLE	(CTRL) P	10						
DC1	(CTRL) Q	11[2]						
DC2	(CTRL) R	12						
DC3	(CTRL) S	13[2]	H					
DC4	(CTRL) T	14						
NAK	(CTRL) U	15						
SYN	(CTRL) V	16						
ETB	(CTRL) W	17						
CAN	(CTRL) X	18		20	98		00	
EM	(CTRL) Y	19						
SUB	(CTRL) Z	1A						
ESC	ESC	1B						
ESC	(CTRL) [	1B						
FS	(CTRL) \	1C						
GS	(CTRL)]	1D						
RS	SHIFT/(CTRL) ^	1E						
US	SHIFT/(CTRL) _	1F						
Space	Space bar	20		20	20		20	
!	!	21		21	21		21	
"	"	22		22	22		22	
#	#	23		23	23		23	
$	$	24		24	24		24	
%	%	25		25	25		25	

Notes:

[1] Not applicable; does not interact with keyboard.

[2] The DSR recognizes DC1 and DC3 but never returns them to the application.

Table B-3. 940 EVT Key Character Code Transformations (Continued)

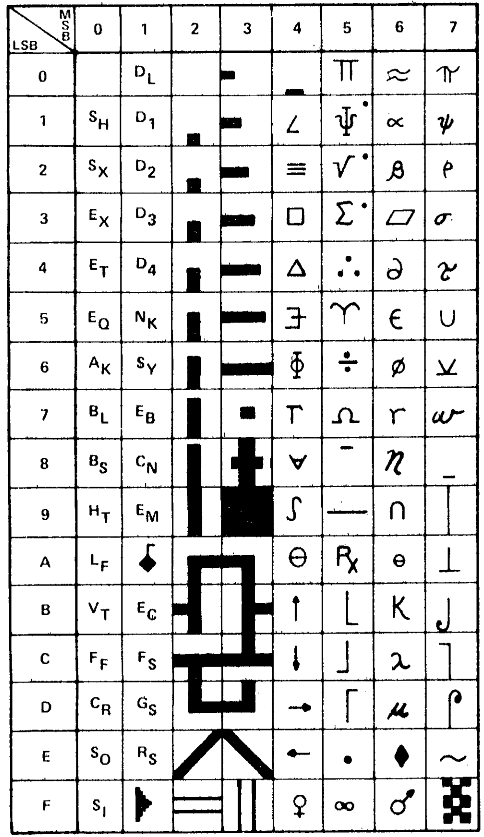
ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
&	&	26		26	26		26	
'	'	27		27	27		27	
(	(	28		28	28		28	
)	)	29		29	29		29	
*	*	2A		2A	2A		2A	
+	+	2B		2B	2B		2B	
,	,	2C		2C	2C		2C	
-	-	2D		2D	2D		2D	
.	.	2E		2E	2E		2E	
/	/	2F		2F	2F		2F	
0	0	30		30	30		30	
1	1	31		31	31		31	
2	2	32		32	32		32	
3	3	33		33	33		33	
4	4	34		34	34		34	
5	5	35		35	35		35	
6	6	36		36	36		36	
7	7	37		37	37		37	
8	8	38		38	38		38	
9	9	39		39	39		39	
:	:	3A		3A	3A		3A	
;	;	3B		3B	3B		3B	
<	<	3C		3C	3C		3C	
=	=	3D		3D	3D		3D	
>	>	3E		3E	3E		3E	
?	?	3F		3F	3F		3F	
@	@	40		40	40		40	
A	A	41		41	41		41	
B	B	42		42	42		42	
C	C	43		43	43		43	
D	D	44		44	44		44	
E	E	45		45	45		45	
F	F	46		46	46		46	
G	G	47		47	47		47	
H	H	48		48	48		48	
I	I	49		49	49		49	
J	J	4A		4A	4A		4A	
K	K	4B		4B	4B		4B	
L	L	4C		4C	4C		4C	
M	M	4D		4D	4D		4D	
N	N	4E		4E	4E		4E	
O	O	4F		4F	4F		4F	
P	P	50		50	50		50	
Q	Q	51		51	51		51	
R	R	52		52	52		52	
S	S	53		53	53		53	
T	T	54		54	54		54	
U	U	55		55	55		55	
V	V	56		56	56		56	

Table B-3. 940 EVT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
W	W	57		57	57		57	
X	X	58		58	58		58	
Y	Y	59		59	59		59	
Z	Z	5A		5A	5A		5A	
[	[	5B		5B	5B		5B	
\	\	5C		5C	5C		5C	
]	]	5D		5D	5D		5D	
^	^	5E		5E	5E		5E	
_	_	5F		5F	5F		5F	
`	`	60		60	60		60	
a	a	61		61	61		61	
b	b	62		62	62		62	
c	c	63		63	63		63	
d	d	64		64	64		64	
e	e	65		65	65		65	
f	f	66		66	66		66	
g	g	67		67	67		67	
h	h	68		68	68		68	
i	i	69		69	69		69	
j	j	6A		6A	6A		6A	
k	k	6B		6B	6B		6B	
l	l	6C		6C	6C		6C	
m	m	6D		6D	6D		6D	
n	n	6E		6E	6E		6E	
o	o	6F		6F	6F		6F	
p	p	70		70	70		70	
q	q	71		71	71		71	
r	r	72		72	72		72	
s	s	73		73	73		73	
t	t	74		74	74		74	
u	u	75		75	75		75	
v	v	76		76	76		76	
w	w	77		77	77		77	
x	x	78		78	78		78	
y	y	79		79	79		79	
z	z	7A		7A	7A		7A	
{	{	7B		7B	7B		7B	
|	|	7C		7C	7C		7C	
}	}	7D		7D	7D		7D	
~	~	7E		7E	7E		7E	
	TAB left	1B32						
	ERASE EOS	1B3D						
	Up arrow	1B41		20	95			
	Down arrow	1B42		20	8A			
	Right arrow	1B43		20	20		20	
	Left arrow	1B44						
	HOME	1B48		20	8C			
	ERASE MSG	1B49						
	ERASE ALL	1B4B		20	8E			

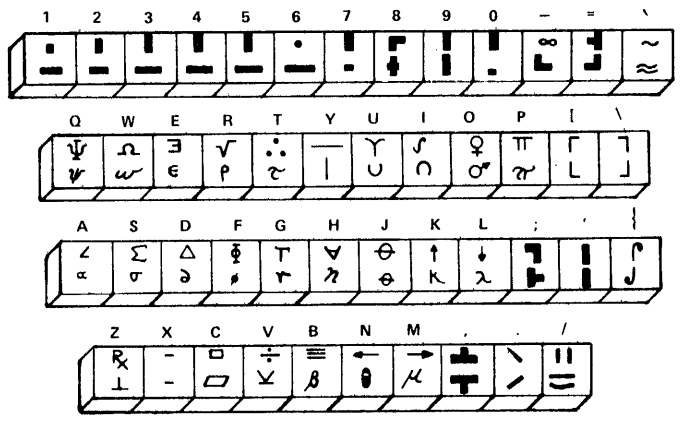
Table B-3. 940 EVT Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
	INS LINE	1B50	S					
	DEL LINE	1B51	S					
	PRINT	1B57	STE	20	99	EVT		
	SCROLL DOWN	1B61						
	PREV REGN	1B63						
	NEXT REGN	1B64						
	PREV PAGE	1B65						
	NEXT PAGE	1B66	STE	20	9B		00	
	PREV FORM	1B67	S					
	NEXT FORM	1B68	S					
	F1	1B6931	E	20	81	EVT		
	F2	1B6932	E	20	82	EVT		
	F3	1B6933	E	20	83	EVT		
	F4	1B6934	E	20	84	EVT		
	F5	1B6935	E	20	85	EVT		
	F6	1B6936	E	20	86	EVT		
	F7	1B6937	E	20	96	EVT		
	F8	1B6938	E	20	97	EVT		
	F9	1B6939	E	20	80	EVT		
	F10	1B693A	E	20	9A	EVT		
	F11	1B693B	E	20	9C	EVT		
	F12	1B693C	E	20	9D	EVT		
	F13	1B693D	E	20	9E	EVT		
	F14	1B693E	E	20	9F	EVT		
	F15	1B693F						
	F16	1B6940						
	F17	1B6941						
	F18	1B6942						
	F19	1B6943						
	F20	1B6944						
	F21	1B6945						
	F22	1B6946						
	F23	1B6947						
	F24	1B6948						
	SKIP left	1B74		20	94			
	SKIP right	1B7330		20	8B			
	SEND	1BE9F1	F	20	93	EOF	00 EOF	



SC2 Character Set

2280966



SC2 Keyboard Layout

Figure B-3. 940 EVT Graphics Characters

Table B-4. Business System Terminal Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
NULL	(CTRL)`	00						N/A[1]
SOH	(CTRL) A	01						
STX	(CTRL) B	02						
ETX	(CTRL) C	03						
EOT	(CTRL) D	04						
ENQ	(CTRL) E	05						
ACK	(CTRL) F	06						
BELL	(CTRL) G	07						
BS	BACK SPACE	08						
BS	(CTRL) H	08						
HT	TAB	09		20	89		00	
HT	(CTRL) I	09		20	89		00	
LF	(CTRL) J	0A		20	87			
VT	(CTRL) K	0B						
FF	(CTRL) L	0C						
CR	(CTRL) M	0D		20	8D		00	
CR	RETURN	0D		20	8D		00	
CR	ENTER	1B6971	F	20	93	EOF	00 EOF	
SO	(CTRL) N	0E						
SI	(CTRL) O	0F						
DLE	(CTRL) P	10						
DC1	(CTRL) Q	11[2]						
DC2	(CTRL) R	12						
DC3	(CTRL) S	13[2]	H					
DC4	(CTRL) T	14						
NAK	(CTRL) U	15						
SYN	(CTRL) V	16						
ETB	(CTRL) W	17						
CAN	(CTRL) X	18		20	98		00	
EM	(CTRL) Y	19						
SUB	(CTRL) Z	1A						
ESC	ESC	1B						
ESC	(CTRL) [	1B						
ESC	(CTRL) {	1B						
FS	(CTRL) \	1C						
GS	CTRL/(SHIFT)]	1D						
RS	CTRL/(SHIFT) 6	1E						
US	CTRL/(SHIFT) _	1F						
Space	Space bar	20		20			20	
!	!	21		21			21	
"	"	22		22			22	
#	#	23		23			23	
$	$	24		24			24	
%	%	25		25			25	

Notes:

[1] Not applicable; does not interact with keyboard.

[2] The DSR recognizes DC1 and DC3 but never returns them to the application.

Table B-4. Business System Terminal Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	ASCII Task Edit					
			Type	Buffer	Event	Flag	ASCII	Direct
&	&	26		26			26	
'	'	27		27			27	
(	(	28		28			28	
)	)	29		29			29	
*	*	2A		2A			2A	
+	+	2B		2B			2B	
,	,	2C		2C			2C	
-	-	2D		2D			2D	
.	.	2E		2E			2E	
/	/	2F		2F			2F	
0	0	30		30			30	
1	1	31		31			31	
2	2	32		32			32	
3	3	33		33			33	
4	4	34		34			34	
5	5	35		35			35	
6	6	36		36			36	
7	7	37		37			37	
8	8	38		38			38	
9	9	39		39			39	
:	:	3A		3A			3A	
;	;	3B		3B			3B	
<	<	3C		3C			3C	
=	=	3D		3D			3D	
>	>	3E		3E			3E	
?	?	3F		3F			3F	
@	@	40		40			40	
A	A	41		41			41	
B	B	42		42			42	
C	C	43		43			43	
D	D	44		44			44	
E	E	45		45			45	
F	F	46		46			46	
G	G	47		47			47	
H	H	48		48			48	
I	I	49		49			49	
J	J	4A		4A			4A	
K	K	4B		4B			4B	
L	L	4C		4C			4C	
M	M	4D		4D			4D	
N	N	4E		4E			4E	
O	O	4F		4F			4F	
P	P	50		50			50	
Q	Q	51		51			51	
R	R	52		52			52	
S	S	53		53			53	
T	T	54		54			54	
U	U	55		55			55	
V	V	56		56			56	

Table B-4. Business System Terminal Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
W	W	57		57			57	
X	X	58		58			58	
Y	Y	59		59			59	
Z	Z	5A		5A			5A	
[	[	5B		5B			5B	
\	\	5C		5C			5C	
]	]	5D		5D			5D	
^	^	5E		5E			5E	
_	_	5F		5F			5F	
`	`	60		60			60	
a	a	61		61			61	
b	b	62		62			62	
c	c	63		63			63	
d	d	64		64			64	
e	e	65		65			65	
f	f	66		66			66	
g	g	67		67			67	
h	h	68		68			68	
i	i	69		69			69	
j	j	6A		6A			6A	
k	k	6B		6B			6B	
l	l	6C		6C			6C	
m	m	6D		6D			6D	
n	n	6E		6E			6E	
o	o	6F		6F			6F	
p	p	70		70			70	
q	q	71		71			71	
r	r	72		72			72	
s	s	73		73			73	
t	t	74		74			74	
u	u	75		75			75	
v	v	76		76			76	
w	w	77		77			77	
x	x	78		78			78	
y	y	79		79			79	
z	z	7A		7A			7A	
{	{	7B		7B			7B	
|	|	7C		7C			7C	
}	}	7D		7D			7D	
~	~	7E		7E			7E	
	Left arrow	1B44						
	Up arrow	1B41		20	95			
	Right arrow	1B43		20	20		20	
	Down arrow	1B42		20	8A			
	(ALT) ERASE INPUT	1B61						
	(ALT) ERASE FIELD	1B62						
	FIELD right	0A		20	87			
	BACK SPACE	08						
	SKIP	1B73		20	8B			

Table B-4. Business System Terminal Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
	FIELD left	1B74		00	94			
	TAB	09		00	89		00	
	(SHIFT) BACK SPACE	1B32						
	DEL CHAR	1B51	S					
	(SHIFT) ESC	1B66	STE	00	9B		00	
	(SHIFT) Blank orange	1B65	S					
	(ALT) CMD	1B64						
	(ALT) Blank orange	1B63						
	CMD	1B68	SE	00	98	EVT		
	Blank orange	1B67						
	HOME	1B48						
	(SHIFT) HOME	1B48		00	8C			
	ERASE FIELD	1B3D						
	(SHIFT) ERASE INPUT	1B4A						
	(SHIFT) ERASE FIELD	1B49						
	(SHIFT) Blank gray	1B3C						
	INS CHAR	1B50	S					
	Blank gray	1B4E		00	8F			
	PRINT		STE	00	99	EVT		
	(ALT) 1[3]	A2						
	(ALT) 4[3]	1B7931		31	31		31	
	(ALT) 2[3]	A4						
	(ALT) 5[3]	1B7932		32	32		32	
	(ALT) 3[3]	A6						
	(ALT) 6[3]	1B7933		33	33		33	
	ERASE INPUT	1B4B		4B	4B			
	F1	1B6931	E	20	81	EVT		
	F2	1B6932	E	20	82	EVT		
	F3	1B6933	E	20	83	EVT		
	F4	1B6934	E	20	84	EVT		
	F5	1B6935	E	20	85	EVT		
	F6	1B6936	E	20	86	EVT		
	F7	1B6937	E	20	96	EVT		
	F8	1B6938	E	20	97	EVT		
	(SHIFT) F1	1B6939	E	20	80	EVT		
	(SHIFT) F2	1B693A	E	20	9A	EVT		
	(SHIFT) F3	1B693B	E	20	9C	EVT		
	(SHIFT) F4	1B693C	E	20	9D	EVT		
	(SHIFT) F5	1B693D	E	20	9E	EVT		
	(SHIFT) F6	1B693E	E	20	9F	EVT		

Note:

[3] The numbers are on the numeric keypad, not across the second row of the keyboard.

Table B-5. 820 KSR Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
NULL	(CTRL) [	00/80	ES	00	80	EVT		00
SOH	(CTRL) A	81/01	EST	00	81	EVT		01
STX	(CTRL) B	82/02	EST	00	82	EVT		02
ETX	(CTRL) C	03/83	EST	00	83	EVT		03
EOT	(CTRL) D	84/04	ES	00	84	EVT		04
ENQ	(CTRL) E	05/85	EST	00	85	EVT		05
ACK	(CTRL) F	06/86	ES	00	86	EVT		06
BELL	(CTRL) G	87/07						07
BS	(CTRL) H	88/08						08
BS	BACKSPACE	88/08						08
HT	(CTRL) I	09/89	ST	00	89		09	09
HT	TAB	09/89	ST	00	89		09	09
LF	(CTRL) J	0A/8A	S	00	8A		0A	0A
LF	LINE FEED	0A/8A	S	00	8A		0A	0A
VT	(CTRL) K	8B/0B	ST	00	8B			0B
FF	(CTRL) L	0C/8C	ST	00	8C			0C
CR	(CTRL) M	8D/0D	S	00	8D		00	0D
CR	RETURN	8D/0D	S	00	8D		00	0D
SO	(CTRL) N	8E/0E	EST	00	8E			0E
SI	(CTRL) O	0F/8F	E	00	8F			0F
DLE	(CTRL) P	90/10						10
DC1	(CTRL) Q	11/91						11
DC2	(CTRL) R	12/92						12
DC3	(CTRL) S	93/13	H					
DC4	(CTRL) T	14/94	T	00	94			14
NAK	(CTRL) U	95/15	T	00	95			15
SYN	(CTRL) V	96/16	E	00	96	EVT		16
ETB	(CTRL) W	17/97	E	00	97	EVT		17
CAN	(CTRL) X	18/98	E	00	98	EVT		18
EM	(CTRL) Y	99/19	EF	00	93	EVTEOF	00 EOF	00
SUB	(CTRL) Z	9A/1A	E	00	9A	EVT		1A
ESC	ESC	1B/9B						1B
FS	(CTRL) \	9C/1C	E	00	9C	EVT		1C
GS	(CTRL) {	1D/9D	E	00	9D	EVT		1D
RS	(CTRL) =	1E/9E	E	00	9E	EVT		1E
US	(CTRL) -	9F/1F	ET	00	9F	EVT		1F
Space	Space bar	A0/20		20			20	20
!	!	21/A1		21			21	21
"	"	22/A2		22			22	22
#	#	A3/23		23			23	23
$	$	24/A4		24			24	24
%	%	A5/25		25			25	25
&	&	A6/26		26			26	26
'	'	27/A7		27			27	27
(	(	28/A8		28			28	28
)	)	A9/29		29			29	29
*	*	AA/2A		2A			2A	2A
+	+	2B/AB		2B			2B	2B

Table B-5. 820 KSR Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	ASCII Task Edit Event	ASCII Task Edit Flag	ASCII	Direct
,	,	AC/2C		2C			2C	2C
-	-	2D/AD		2D			2D	2D
.	.	2E/AE		2E			2E	2E
/	/	AF/2F		2F			2F	2F
0	0	30/B0		30			30	30
1	1	B1/31		31			31	31
2	2	B2/32		32			32	32
3	3	33/B3		33			33	33
4	4	B4/34		34			34	34
5	5	35/B5		35			35	35
6	6	36/B6		36			36	36
7	7	B7/37		37			37	37
8	8	B8/38		38			38	38
9	9	39/B9		39			39	39
:	:	3A/BA		3A			3A	3A
;	;	BB/3B		3B			3B	3B
<	<	3C/BC		3C			3C	3C
=	=	BD/3D		3D			3D	3D
>	>	BE/3E		3E			3E	3E
?	?	3F/BF		3F			3F	3F
@	@	C0/40		40			40	40
A	A	41/C1		41			41	41
B	B	42/C2		42			42	42
C	C	C3/43		43			43	43
D	D	44/C4		44			44	44
E	E	C5/45		45			45	45
F	F	C6/46		46			46	46
G	G	47/C7		47			47	47
H	H	48/C8		48			48	48
I	I	C9/49		49			49	49
J	J	CA/4A		4A			4A	4A
K	K	4B/CB		4B			4B	4B
L	L	CC/4C		4C			4C	4C
M	M	4D/CD		4D			4D	4D
N	N	4E/CE		4E			4E	4E
O	O	CF/4F		4F			4F	4F
P	P	50/D0		50			50	50
Q	Q	D1/51		51			51	51
R	R	D2/52		52			52	52
S	S	53/D3		53			53	53
T	T	D4/54		54			54	54
U	U	55/D5		55			55	55
V	V	56/D6		56			56	56
W	W	D7/57		57			57	57
X	X	D8/58		58			58	58
Y	Y	59/D9		59			59	59
Z	Z	5A/DA		5A			5A	5A
[	[	DB/5B		5B			5B	5B
\	\	5C/DC		5C			5C	5C

Table B-5. 820 KSR Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
]	]	DD/5D		5D			5D	5D
^	^	DE/5E		5E			5E	5E
_	_	5F/DF		5F			5F	5F
`	`	60/E0		60			60	60
a	a	E1/61		61			61	61
b	b	E2/62		62			62	62
c	c	63/E3		63			63	63
d	d	E4/64		64			64	64
e	e	65/E5		65			65	65
f	f	66/E6		66			66	66
g	g	E7/67		67			67	67
h	h	E8/68		68			68	68
i	i	69/E9		69			69	69
j	j	6A/EA		6A			6A	6A
k	k	EB/6B		6B			6B	6B
l	l	6C/EC		6C			6C	6C
m	m	ED/6D		6D			6D	6D
n	n	EE/6E		6E			6E	6E
o	o	6F/EF		6F			6F	6F
p	p	F0/70		70			70	70
q	q	71/F1		71			71	71
r	r	72/F2		72			72	72
s	s	F3/73		73			73	73
t	t	74/F4		74			74	74
u	u	F5/75		75			75	75
v	v	F6/76		76			76	76
w	w	77/F7		77			77	77
x	x	78/F8		78			78	78
y	y	F9/79		79			79	79
z	z	FA/7A		7A			7A	7A
{	{	7B/FB		7B			7B	7B
¦	¦	FC/7C		7C			7C	7C
}	}	7D/FD		7D			7D	7D
~	~	7E/FE		7E			7E	7E
DEL	DEL	00/7F						
ENTER	ENTER*							

Note:

* This key is user programmable.

Table B-6. 783 TPD Key Character Code Transformations

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit				
				Buffer	Event	Flag	ASCII	Direct
NULL	(CTRL) [	00/80	SE	00	80	EVT		00
SOH	(CTRL) A	81/01	STE	00	81	EVT		01
STX	(CTRL) B	82/02	STE	00	82	EVT		02
ETX	(CTRL) C	03/83	STE	00	83	EVT		03
EOT	(CTRL) D	84/04	SE	00	84	EVT		04
ENQ	(CTRL) E	05/85	STE	00	85	EVT		05
ACK	(CTRL) F	06/86	SE	00	86	EVT		06
BELL	(CTRL) G	87/07						07
BS	(CTRL) H	88/08						08
HT	(CTRL) I	09/89	ST	00	89		09	09
LF	(CTRL) J	0A/8A	S	00	8A		0A	0A
LF	LINE FEED	0A/8A	S	00	8A		0A	0A
VT	(CTRL) K	8B/0B	ST	00	8B			0B
FF	(CTRL) L	0C/8C	ST	00	8C			0C
CR	(CTRL) M	8D/0D	S	00	8D		00	0D
CR	RETURN	8D/0D	S	00	8D		00	0D
CR	ENTER	8D/0D	S	00	8D		00	0D
SO	(CTRL) N	8E/0E	STE	00	8E			0E
SI	(CTRL) O	0F/8F	E	00	8F			0F
DLE	(CTRL) P	90/10						10
DC1	(CTRL) Q	11/91						11
DC2	(CTRL) R	12/92						12
DC3	(CTRL) S	93/13	H					
DC4	(CTRL) T	14/84	T	00	94			14
NAK	(CTRL) U	95/15	T	00	95			15
SYN	(CTRL) V	96/16	E	00	96	EVT		16
ETB	(CTRL) W	17/97	E	00	97	EVT		17
CAN	(CTRL) X	18/98	E	00	98	EVT		18
EM	(CTRL) Y	99/19	EF	00	93	EVTEOF	00 EOF	00
SUB	(CTRL) Z	9A/1A	E	00	9A	EVT		1A
ESC	ESC	1B/9B						1B
FS	(CTRL) ,	9C/1C	E	00	9C	EVT		1C
GS	(CTRL) {	1D/9D	E	00	9D	EVT		1D
RS	(CTRL) .	1E/9D	E	00	9E	EVT		1E
US	(CTRL) /	9F/1F	ET	00	9F	EVT		1F
Space	Space bar	A0/20		20			20	20
!	!	21/A1		21			21	21
"	"	22/A2		22			22	22
#	#	A3/23		23			23	23
$	$	24/A4		24			24	24
%	%	A5/25		25			25	25
&	&	A6/26		26			26	26
'	'	27/A7		27			27	27
(	(	28/A8		28			28	28
)	)	A9/29		29			29	29
*	*	AA/2A		2A			2A	2A
+	+	2B/AB		2B			2B	2B
,	,	AC/2C		2C			2C	2C

Table B-6. 783 TPD Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
-	-	2D/AD		2D			2D	2D
.	.	2E/AE		2E			2E	2E
/	/	AF/2F		2F			2F	2F
0	0	30/B0		30			30	30
1	1	B1/31		31			31	31
2	2	B2/32		32			32	32
3	3	33/B3		33			33	33
4	4	B4/34		34			34	34
5	5	35/B5		35			35	35
6	6	36/B6		36			36	36
7	7	B7/37		37			37	37
8	8	B8/38		38			38	38
9	9	39/B9		39			39	39
:	:	3A/BA		3A			3A	3A
;	;	BB/3B		3B			3B	3B
<	<	3C/BC		3C			3C	3C
=	=	BD/3D		3D			3D	3D
>	>	BE/3E		3E			3E	3E
?	?	3F/BF		3F			3F	3F
@	@	C0/40		40			40	40
A	A	41/C1		41			41	41
B	B	42/C2		42			42	42
C	C	C3/43		43			43	43
D	D	44/C4		44			44	44
E	E	C5/45		45			45	45
F	F	C6/46		46			46	46
G	G	47/C7		47			47	47
H	H	48/C8		48			48	48
I	I	C9/49		49			49	49
J	J	CA/4A		4A			4A	4A
K	K	4B/CB		4B			4B	4B
L	L	CC/4C		4C			4C	4C
M	M	4D/CD		4D			4D	4D
N	N	4E/CE		4E			4E	4E
O	O	CF/4F		4F			4F	4F
P	P	50/D0		50			50	50
Q	Q	D1/51		51			51	51
R	R	D2/52		52			52	52
S	S	53/D3		53			53	53
T	T	D4/54		54			54	54
U	U	55/D5		55			55	55
V	V	56/D6		56			56	56
W	W	D7/57		57			57	57
X	X	D8/58		58			58	58
Y	Y	59/D9		59			59	59
Z	Z	5A/DA		5A			5A	5A
[	[	DB/5B		5B			5B	5B
\	(CTRL) 7	5C/DC		5C			5C	5C
]	]	DD/5D		5D			5D	5D

Table B-6. 783 TPD Key Character Code Transformations (Continued)

ASCII Character	Terminal Key	Terminal Generated Code	Type	ASCII Task Edit Buffer	Event	Flag	ASCII	Direct
^	^	DE/5E		5E			5E	5E
_	_	5F/DF		5F			5F	5F
`	(CTRL) 0	60/E0		60			60	60
a	a	E1/61		61			61	61
b	b	E2/62		62			62	62
c	c	63/E3		63			63	63
d	d	E4/64		64			64	64
e	e	65/E5		65			65	65
f	f	66/E6		66			66	66
g	g	E7/67		66			67	67
h	h	E8/68		68			68	68
i	i	69/E9		69			69	69
j	j	6A/EA		6A			6A	6A
k	k	EB/6B		6B			6B	6B
l	l	6C/EC		6C			6C	6C
m	m	ED/6D		6D			6D	6D
n	n	EE/6E		6E			6E	6E
o	o	6F/EF		6F			6F	6F
p	p	F0/70		70			70	70
q	q	71/F1		71			71	71
r	r	72/F2		72			72	72
s	s	F3/73		73			73	73
t	t	74/F4		74			74	74
u	u	F5/75		75			75	75
v	v	F6/76		76			76	76
w	w	77/F7		77			77	77
x	x	78/F8		78			78	78
y	y	F9/79		79			79	79
z	z	FA/7A		7A			7A	7A
{	{	7B/FB		7B			7B	7B
¦	(CTRL) 8	FC/7C		7C			7C	7C
}	}	7D/FD		7D			7D	7D
~	(CTRL) 9	7E/FE		7E			7E	7E
DEL	DEL	00/7F						

B.3 733 AND 743 DATA TERMINALS SYSGENED AS DEVICE TYPES ASR OR KSR

Table B-8 gives the character set of the 733 and 743 data terminals followed by an explanation of the characters that perform special functions on these terminals.

Table B-7. Display Terminal Graphics Character Sets

CODE	CHARACTER
00	*
01	*
02	*
03	*
04	*
05	*
06	*
07	*
08	*
09	*
0A	*
0B	*
0C	*
0D	*
0E	*
0F	*
10	*
11	*
12	*
13	*
14	*
15	*
16	*
17	*
18	*
19	*
1A	*
1B	*
1C	*
1D	*
1E	*
1F	*

2279759

Table B-8. 733 and 743 Data Terminal Character Set

Character	Hexadecimal	Character	Hexadecimal	Character	Hexadecimal
Space	20	@	40	'	60
!	21	A	41	a	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
,	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
-	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	[	5B	{	7B
<	3C	/	5C	¦	7C
=	3D	]	5D	}	7D
>	3E	^	5E	~	7E
?	3F	_	5F		

Notes:

The Keyboard:

BS character (>08) is returned to printer as LF and BS. Deletes most recently entered character in buffer.

HT character (>09) is returned to the printer as space. Character is placed in buffer.

LF character (>0A) is returned to printer as LF, but is not stored in buffer.

CR character (>0D) is returned to printer as CR, and character is not placed in buffer. Character terminates the record.

DC3 character (>13) is not stored in buffer. When DC3 is first character of record, the record is an end-of-file record.

ESC character (>1B), when entered during output, terminates output with a write error.

Table B-8. 733 and 743 Data Terminal Character Set (Continued)

Notes (Continued):

The Keyboard: (continued)

DEL character (>7F) is returned to printer as a line feed and carriage return. Character deletes current input record.

Maximum buffer size is 83 characters.

The Printer:

BS character (>08) results in a backspace operation.

HT character (>09) results in printing a space.

LF character (>0A) results in a line feed operation.

CR character (>0D) results in a carriage return operation.

End-of-record occurs when specified number of characters have been printed.

Maximum buffer size is 83 characters.

Cassette Input:

The characters LF (>0A) and DEL (>7F) or the character DEL at the beginning of a record are ignored. The first valid character of the record is the character following the DEL.

The characters HT (>09), FF (>0C), BEL (>07), and BS (>08) are stored in the user's buffer.

The character ETB (>17) is stored in the user's buffer as a CR (>0D).

The character DC3 (>13) as the first character of a record indicates an end-of-file record. The system returns end-of-file status after positioning the tape at the beginning of the next record. The DC3 is not stored in the buffer.

An EOF may be added to a cassette by the following procedure:

a. Position the cassette for recording the EOF record.

b. With the cassette drive in the local record mode, type the character sequence DC3 (CTRL/S), RETURN, and NEW LINE on the keyboard.

c. Press the RECORD OFF button.

d. Rewind the cassette.

The character CR (>0D) indicates end-of-record, and is not stored in the buffer.

Maximum buffer size is 83 characters for U.S. and European terminals, and 80 characters for Katakana terminals.

Cassette Output:

The end-of-block character sequence is CR (>0D) LF (>0A) DC4 (>14) DEL (>7F). The end-of-file character sequence is DC3 (>13) CR DC4 DEL. These characters are supplied by the system, not by the user.

The characters HT (>09), FF (>0C), BEL (>07), and BS (>08) are written unchanged.

The character CR (>0D) is translated to ETB (>17) and written.

The character DC3 (>13) may be placed within a record, but may not be the first character of a record other than the end-of-file record.

Table B-8. 733 and 743 Data Terminal Character Set (Continued)

Notes (Continued):

Cassette Output: (continued)	End-of-record occurs when specified number of characters have been written. Maximum buffer size is 83 characters for U.S. and European terminals, and 80 characters for Katakana terminals.

B.3.1 End-of-File Sequence

The end-of-file sequence for the 733 and 743 data terminals is a DC3 character.

B.3.2 End-of-Record Sequence

The end-of-record sequence for the 733 and 743 data terminals is a CR character.

B.4 CARD READER

Table B-9 provides a hexadecimal-to-row-punch conversion table.

Table B-9. Card Reader Character Set

Hexadecimal	Row Punches	Hexadecimal	Row Punches
00	12-0-9-8-1	2F	0-1
01	12-9-1	30	0
02	12-9-2	31	1
03	12-9-3	32	2
04	9-7	33	3
05	0-9-8-5	34	4
06	0-9-8-6	35	5
07	0-9-8-7	36	6
08	11-9-6	37	7
09	12-9-5	38	8
0A	0-9-5	39	9
0B	12-9-8-3	3A	8-2
0C	12-9-8-4	3B	11-8-6
0D	12-9-8-5	3C	12-8-4
0E	12-9-8-6	3D	8-6
0F	12-9-8-7	3E	0-8-6
10	12-11-9-8-1	3F	0-8-7
11	11-9-1	40	8-4
12	11-9-2	41	12-1
13	11-9-3	42	12-2
14	9-8-4	43	12-3
15	9-8-5	44	12-4
16	9-2	45	12-5
17	0-9-6	46	12-6
18	11-9-8	47	12-7
19	11-9-8-1	48	12-8
1A	9-8-7	49	12-9
1B	0-9-7	4A	11-1
1C	11-9-8-4	4B	11-2
1D	11-9-8-5	4C	11-3
1E	11-9-8-6	4D	11-4
1F	11-9-8-7	4E	11-5
20	None	4F	11-6
21	12-8-7	50	11-7
22	8-7	51	11-8
23	8-3	52	11-9
24	11-8-3	53	0-2
25	0-8-4	54	0-3
26	12	55	0-4
27	8-5	56	0-5
28	12-8-5	57	0-6
29	11-8-5	58	0-7
2A	11-8-4	59	0-8
2B	12-8-6	5A	0-9
2C	0-8-3	5B	12-8-2
2D	11	5C	0-8-2
2E	12-8-3	5D	11-8-2

Table B-9. Card Reader Character Set (Continued)

Hexadecimal	Row Punches	Hexadecimal	Row Punches
5E	11-8-7	6F	12-11-6
5F	0-8-5	70	12-11-7
60	8-1	71	12-11-8
61	12-0-1	72	12-11-9
62	12-0-2	73	11-0-2
63	12-0-3	74	11-0-3
64	12-0-4	75	11-0-4
65	12-0-5	76	11-0-5
66	12-0-6	77	11-0-6
67	12-0-7	78	11-0-7
68	12-0-8	79	11-0-8
69	12-0-9	7A	11-0-9
6A	12-11-1	7B	12-0
6B	12-11-2	7C	12-11
6C	12-11-3	7D	11-0
6D	12-11-4	7E	11-0-1
6E	12-11-5	7F	12-9-7

Notes:

End of record occurs when the specified number of characters, or 80 characters have been read.

Maximum buffer size is 80 characters.

B.4.1 End-of-File Sequence

The end-of-file sequence for the card reader is a slash (/) in column one and an asterisk (*) in column two.

B.5 LINE PRINTER

DX10 provides support for various types of line printers, including line printers supporting the eight-bit Katakana code. Table B-10 lists the characters which have special significance for the line printer.

Table B-10. Line Printer Character Set

Character	Hexadecimal	Character	Hexadecimal	Character	Hexadecimal
Space	20	@	40	'	60
!	21	A	41	a	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
'	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
-	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	[	5B	{	7B
<	3C	/	5C	¦	7C
=	3D	]	5D	}	7D
>	3E	^	5E	~	7E
?	3F	_	5F	DEL	7F

Notes:

The character BS (>08) results in a backspace operation.

The character HT (>08) results in a single space.

The character LF (>0A) results in a line feed operation.

The character CR (>0D) results in a carriage return operation.

The character FF (>0C) results in a form feed operation.

The character BEL (>07) results in a tone signal.

For Model 306, Model 810, and Model 588 Line Printers, the character S0 (>0E) results in character elongation for the line of characters following the S0. Elongation doubles the width of the characters, and a line of elongated characters contains one-half the number of characters on a normal line; i.e., 40 or 66. 50 is ignored by 2230 and 2260 line printers.

B.5.1 810 Line Printer Control Characters

The following control characters can be sent using the write direct call:

VT (Vertical Tab) causes data, if any, in the line buffer to be printed and advances the paper to the next vertical tab location or top of form, whichever occurs first. If no vertical tabs are set, a VT command causes the paper to be advanced to top of form.

HT (Horizontal Tab) causes spaces to be entered in the line buffer up to the next horizontal tab location, where printing will begin.

DC1 (Select) selects the printer, enabling it to receive data. The controller responds to this control character but does not transmit a verification.

DC2 + n (Tab to Line) causes the paper drive system to advance to the line specified by n. The value n must be greater than the present line.

DC3 (Does not select) causes the printer to go offline and any subsequent characters transmitted to be discarded until a DC1 character is received. The printer does not go offline until an LF character is received after the DC3; any characters received before the DC3 are printed. The controller responds to this control character but does not transmit a verification.

DEL (Delete) clears the line buffer.

NUL (Null) terminates the tab setting sequence (see below); otherwise it is ignored.

DC4 + n (Tab to Address) causes the carriage to advance at high speed to the column specified by n. The value n must be greater than the present carriage position. If n is less than the present carriage position, this command is ignored.

$ESC + 1 + n_1 + n_2 + ... + n_k + NUL$ (Set Vertical Tabs) clears all existing vertical tabs and sets new tabs at lines n_1, n_2,..., and n_k.

ESC + 2 + n (Set Form Length) sets the form length used by the Form Feed (FF) command to n. Lines per form is set to a default value of 66 at power up.

$ESC + 3 + n_1 + n_2 + ... + n_k + NUL$ (Set Horizontal Tabs) clears existing horizontal tabs and sets new tabs at locations n_1, n_2,..., and n_k.

ESC + 4 sets paper drive system to 6 lines per inch.

ESC + 5 sets paper drive system to 8 lines per inch.

ESC + 6 sets carriage system to 10 characters per inch.

ESC + 7 sets carriage system to 16.5 characters per inch.

ESC + 8 enters parameters into the VFC or VCO channel.

ESC + 9 retrieves parameters from the VFC or VCO channel.

ESC + ; sets the line length to 132 characters.

ESC + : + n sets the line length to n characters (where n equals 2 through 126).

NOTE

> The values n, n_1, n_2, and so on used in the ESC commands represent seven-bit binary numbers. If the parity option is selected on the printer, correct parity must also be supplied.

A horizontal or vertical control character sequence affects only the lines following the control character sequence.

B.6 KIF COLLATING SEQUENCE

A description of the collating sequences for the various country codes supported by DX10 follows:

France/Belgium	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
France Word Processing	ABCDEFGHIJKLMNOPQRSTUVWXYZ aàbcçdeéèfghijklmnopqrstuùvwxyz
Germany/Austria	AÄBCDEFGHIJKLMNOÖPQRSTUÜVWXYZ aäbcdefghijklmnoöpqrsβtuüvwxyz
Japan (Katakana)	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz ｦｧｨｩｪｫｬｭｮｯｰｱ ｲｳｴｵｶｷｸｹｺｻｼｽ ｾｿﾀﾁﾂﾃﾄﾅﾆﾇﾈﾉ ﾊﾋﾌﾍﾎﾏﾐﾑﾒﾓﾔﾕ ﾖﾗﾘﾙﾚﾛﾜﾝﾞﾟ
Norway/Denmark	ABCDEFGHIJKLMNOPQRSTUVWXYZ ÆØÅ abcdefghijklmnopqrstuvwxyz æøå
Sweden/Finland	ABCDEFGHIJKLMNOPQRSTUVWXYÜZÅÄÖ abcdefghijklmnopqrstuvwxyüzåäö
Spanish-speaking countries	ABCDEFGHIJKLMNÑOPQRSTUVWXYZ abcçdefghijklmnñopqrstuvwxyz
Switzerland	ABCDEFGHIJKLMNOPQRSTUVWXYZ aàäbcçdeéèfghijklmnoöpqrstuüvwxyz
United Kingdom	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

2283037

Appendix C

Task State Codes

Several supervisor calls, including all I/O SVCs, return a task state code in byte 1 of the SVC call block. The possible codes are shown in the following list. To view the state of tasks active in the system, use the STS SCI command.

Table C-1. Task State Codes

Code (Hexadecimal)	Significance
00	Active task
01	Active task
04	Terminated task
05	Task in time delay
06	Suspended task
07	Currently executing task
08	Task awaiting character input
09	Task awaiting completion of I/O
0A	Task awaiting assignment of a device for I/O
0B	Task awaiting disk file utility services
0C	Task awaiting compress services
0D	Task awaiting file management services
0E	Task awaiting overlay loader services
0F	Task awaiting initial load
11	Task awaiting disk management services
12	Task awaiting tape management services
13	Task awaiting system overlay loader services
14	Task awaiting task driven supervisor call processor
15	Task awaiting Get Memory services
17	Task suspended for activation of a called task
18	Task awaiting termination task or diagnostic services
19	Task awaiting completion of any I/O
1A	Task awaiting memory management services
1B	Task is eligible for roll-out when requested I/O completes
1C	Task activated while roll is in progress
1D	Suspended for initiate I/O threshold
1E	Suspended for locked directory
1F	Suspended for task management directory buffer
24	Task suspended for queue input
FF	Dummy task state

Appendix D

Reentrant Programming Example in Assembly Language

The following assembly listings show two task segments and two procedure segments. The task segments consist of data (some of it modifiable) while the procedure segments consist of pure code (executable code and constant data).

These four segments (TASK1, TASK2, PROC1, PROC2) were assembled using four separate assemblies (using the macro assembler SDSMAC).

```
TASK1          SDSMAC 947075 *E   08:37:34 TUESDAY, AUG 02, 19XX
                                                            PAGE 0001

 0001                          IDT 'TASK1'
 0002                   *
 0003                   *      TASK HEADER
 0004                   *
 0005                          REF  PROC1
 0006 0000 0078'               DATA TWS,PROC1,0        TRANSFER VECTOR
      0002 0000
      0004 0000
 0007                          DEF  ALUNO,ALNUM,OPEN,OLNUM
 0008                          DEF  WRITE1,W1LNUM,WRITE2,W2LNUM
 0009                          DEF  CLOSE,CLNUM,RELEAS,RLNUM
 0010                   *
 0011                   *      ASSIGN LUNO BLOCK
 0012                   *
 0013 0006                     EVEN
 0014 0006    00        ALUNO  BYTE 0,0                IO,STATUS
      0007    00
 0015 0008    91               BYTE >91                ALUNO OPCODE
 0016 0009    00        ALNUM  BYTE >00                LUNO NUMBER
 0017 000A 0000                DATA 0,0,0,0,0,0,       NOT USED
      000C 0000
      000E 0000
      0010 0000
      0012 0000
      0014 0000
 0018 0016    00        ALUFLG BYTE >04,0              UTILITY FLAGS-GENERATE LUNO
      0017    00
 0019 0018 0000                DATA 0,0,DEVNAM,0,0,0,0,0,0
      001A 0000
      001C 00B5
      001E 0000
      0020 0000
      0022 0000
      0024 0000
      0026 0000
      0028 0000
 0020                   *
 0021                   *      OPEN PRB
 0022                   *
 0023 002A                     EVEN
 0024 002A    00        OPEN   BYTE 0,0                IO,STATUS
      002B    00
 0025 002C    00               BYTE 0                  OPEN
 0026 002D    00        OLNUM  BYTE 0                  LUNO, TO BE FILLED IN
```

```
0027 002E    00          BYTE 0,0                 SYSTEM, USER FLAGS
     002F    00
0028 0030    00          BYTE 0,0                 DEVICE TYPE
     0031    00
0029 0032  0000          DATA 0,0,0
     0034  0000
     0036  0000
0030                *
0031                *    WRITE ONE PRB
0032                *
0033 0038                EVEN
0034 0038    00   WRITE1 BYTE 0,0                 IO,STATUS
     0039    00
0035 003A    0B          BYTE >B                  WRITE ASCII
0036 003B    00   WILNUM BYTE 0                   LUNO,TO BE FILLED IN
0037 003C    00          BYTE 0,0                 SYSTEM,USER FLAGS
TASK2        SDSMAC 947075 *E    08:37:34 TUESDAY, AUG 02, 19XX
     003D    00                                                      PAGE 0002
0038 003E  0098'         DATA TEXT1               BUFFER ADDRESS
0039 0040  0010          DATA TEXT1L              CHAR COUNT
0040 0042  0010          DATA TEXT1L              CHAR COUNT
0041 0044  0000          DATA 0
0042                *
0043                *    WRITE TWO PRB
0044                *
0045 0046                EVEN
0046 0046    00   WRITE2 BYTE 0,0                 IO,STATUS
     0047    00
0047 0048    0B          BYTE >B                  WRITE ASCII
0048 0049    00   W2LNUM BYTE 0                   LUNO, TO BE FILLED IN
0049 004A    00          BYTE 0,0                 SYSTEM, USER FLAGS
     004B    00
0050 004C  00A8          DATA TEXT2               BUFFER ADDRESS
0051 004E  000D          DATA TEXT2L              CHAR COUNT
0052 0050  000D          DATA TEXT2L              CHAR COUNT
0053 0052  0000          DATA 0
0054                *
0055                *    CLOSE PRB
0056                *
0057 0054                EVEN
0058 0054    00   CLOSE  BYTE 0,0                 IO,STATUS
     0055    00
0059 0056    01          BYTE >1                  CLOSE
0060 0057    00   CLNUM  BYTE 0                   LUNO, TO BE FILLED IN
0061 0058    00          BYTE 0,0                 SYSTEM, USER FLAGS
     0059    00
0062 005A  0000          DATA 0,0,0,0             NOT USED
     005C  0000
     005E  0000
     0060  0000
0063                *
0064                *    RELEASE LUNO
0065                *
0066 0062                EVEN
0067 0062    00   RELEAS BYTE 0,0                 IO,STATUS
     0063    00
0068 0064    93          BYTE >93                 RELEASE LUNO
0069 0065    00   RLNUM  BYTE 0                   LUNO, TO BE FILLED IN
```

```
0070 0066 0000          DATA 0,0,0,0,0,0,0,0,0,0 NOT USED
     0068 0000
     006A 0000
     006C 0000
     006E 0000
     0070 0000
     0072 0000
     0074 0000
     0076 0000
0071                *
0072                *       TASK 1 DATA
0073                *
0074 0078           TWS     BSS  16*2
0075 0098   49      TEXT1   TEXT 'I AM TASK ONE.55'
     0099   20
     009A   41
     009B   4D
     009C   20
     009D   54

TASK2           SDSMAC 947075 *E       08:37:34 TUESDAY, AUG 02, 19XX

     009E   41                                                        PAGE 0003
     009F   53
     00A0   4B
     00A1   20
     00A2   4F
     00A3   4E
     00A4   45
     00A5   2E
     00A6   35
     00A7   35
0076      000D      TEXT1L EQU  $-TEXT1
0077 00A8   4E      TEXT2  TEXT 'NOW I AM DONE'.
     00A9   4F
     00AA   57
     00AB   20
     00AC   49
     00AD   20
     00AE   41
     00AF   4D
     00B0   20
     00B1   44
     00B2   4F
     00B3   4E
     00B4   45
0078      000D      TEXT2L EQU  $-TEXT2
0079 00B5   04      DEVNAM BYTE >4
0080 00B6   4C             TEXT 'LP01'
     00B7   50
     00B8   30
     00B9   31
0081                       END
NO ERRORS
```

```
TASK2           SDSMAC 947075 *E    08:40:17 TUESDAY, AUG 02, 19XX
                                                                    PAGE 0001
 0001                          IDT    'TASK2'
 0002                   *
 0003                   *      TASK   HEADER
 0004                   *
 0005                          REF    PROC1
 0006 0000 0078'               DATA   TWS,PROC1,0         TRANSFER VECTOR
      0002 0000
      0004 0000
 0007                          DEF    ALUNO,ALNUM,OPEN,OLNUM
 0008                          DEF    WRITE1,W1LNUM,WRITE2,W2LNUM
 0009                          DEF    CLOSE,CLNUM,RELEASE,RLNUM
 0010                   *
 0011                   *      ASSIGN LUNO BLOCK
 0012                   *
 0013 0006                     EVEN
 0014 0006   00         ALUNO  BYTE 0,0               IO,STATUS
      0007   00
 0015 0008   91                BYTE >91               ALUNO OPCODE
 0016 0009   00         ALNUM  BYTE >00               LUNO NUMBER
 0017 000A 0000                DATA 0,0,0,0,0,0       NOT USED
      000C 0000
      000E 0000
      0010 0000
      0012 0000
      0014 0000
 0018 0016   04         ALUFLG BYTE >04,0             UTILITY FLAGS-GENERATE LUNO
      0017   00
 0019 0018 0000                DATA 0,0,DEVNAM,0,0,0,0,0,0
      001A 0000
      001C 00B5
      001E 0000
      0020 0000
      0022 0000
      0024 0000
      0026 0000
      0028 0000
 0020                   *
 0021                   *      OPEN PRB
 0022                   *
 0023 002A                     EVEN
 0024 002A   00         OPEN   BYTE 0,0               IO,STATUS
      002B   00
 0025 002C   00                BYTE 0                 OPEN
 0026 002D   00         OLNUM  BYTE 0                 LUNO, TO BE FILLED IN
 0027 002E   00                BYTE 0,0               SYSTEM, USER FLAGS
      002F   00
 0028 0030   00                BYTE 0,0               DEVICE TYPE
      0031   00
 0029 0032 0000                DATA 0,0,0
      0034 0000
      0036 0000
 0030                   *
 0031                   *      WRITE ONE PRB
 0032                   *
 0033 0038                     EVEN
 0034 0038              WRITE1 BYTE 0,0               IO,STATUS
      0039   00
 0035 003A   0B                BYTE >B                WRITE ASCII
 0036 003B   00         W1LNUM BYTE 0                 LUNO, TO BE FILLED IN
 0037 003C   00                BYTE 0,0               SYSTEM, USER FLAGS
```

```
TASK2          SDSMAC 947075 *E    08:40:17 TUESDAY, AUG 02, 19XX
                                                              PAGE 0002
     003D   00
0038 003E 0098'         DATA TEXT1              UFFER ADDRESS
0039 0040 0010          DATA TEXT1L             CHAR COUNT
0040 0042 0010          DATA TEXT1L             CHAR COUNT
0041 0044 0000          DATA 0
0042               *
0043               *    WRITE TWO PRB
0044               *
0045 0046               EVEN
0046 0046   00     WRITE2 BYTE 0,0              IO,STATUS
     0047   00
0047 0048   0B          BYTE >B                 WRITE ASCII
0048 0049   00     W2LNUM BYTE 0                LUNO, TO BE FILLED IN
0049 004A   00          BYTE 0,0                SYSTEM, USER FLAGS
     004B   00
0050 004C 00A8          DATA TEXT2              BUFFER ADDRESS
0051 004E 0010          DATA TEXT2L             CHAR COUNT
0052 0050 0010          DATA TEXT2L             CHAR COUNT
0053 0052 0000          DATA 0
0054               *
0055               *    CLOSE PRB
0056               *
0057 0054               EVEN
0058 0054   00     CLOSE  BYTE 0,0              IO,STATUS
     0055   00
0059 0056   01          BYTE >1                 CLOSE
0060 0057   00     CLNUM  BYTE 0                LUNO, TO BE FILLED IN
0061 0058   00          BYTE 0,0                SYSTEM, USER FLAGS
     0059   00
0062 005A 0000          DATA 0,0,0,0            NOT USED
     005C 0000
     005E 0000
     0060 0000
0063               *
0064               *    RELEASE LUNO
0065               *
0066 0062               EVEN
0067 0062   00     RELEAS BYTE 0,0              IO,STATUS
     0063   00
0068 0064   93          BYTE >93                RELEASE LUNO
0069 0065   00     RLNUM  BYTE 0                LUNO, TO BE FILLED IN
0070 0066 0000          DATA 0,0,0,0,0,0,0,0,0 NOT USED
     0068 0000
     006A 0000
     006C 0000
     006E 0000
     0070 0000
     0072 0000
     0074 0000
     0076 0000
0071               *
0072               *    TASK 2 DATA
0073               *
0074 0078          TWS    BSS  16*2
0075 0098   19     TEXT1  TEXT 'I AM TASK TWO.55'
     0099   20
     009A   41
     009B   4D
     009C   20
     009D   54
```

```
TASK2          SDSMAC 947075 *E      08:40:17 TUESDAY, AUG 02, 19XX
                                                                    PAGE 0003
      009E    41
      009F    53
      00A0    4B
      00A1    20
      00A2    54
      00A3    57
      00A4    4F
      00A5    2E
      00A6    35
      00A7    35
 0076       000D    TEXT1L EQU   $-TEXT1
 0077 00A8    4E    TEXT2  TEXT  'NOW I AM THROUGH'.
      00A9    4F
      00AA    57
      00AB    20
      00AC    49
      00AD    20
      00AE    41
      00AF    4D
      00B0    20
      00B1    54
      00B2    48
      00B3    52
      00B4    4F
      00B5    55
      00B6    47
      00B7    48
 0078       0010    TEXT2L EQU   $-TEXT2
 0079 00B8    04    DEVNAM BYTE  >4
      00B9    4C           TEXT  'LP01'
      00BA    50
      00BB    30
      00BC    31
 0081               END
PROC1          SDSMAC 947075 *E    13:13:52 MONDAY, AUG 01, 19XX
                                                               PAGE 0001
 0001               *
 0002               *      PROCEDURE SEGMENT 1
 0003               *
 0004                      IDT   'PROC1'
 0005                      DEF   PROC1
 0006                      REF   PROC2
 0007                      REF   ALUNO,ALNUM,OPEN,OLNUM
 0008                      REF   WRITE1,W1LNUM,WRITE2,W2LNUM
 0009                      REF   CLOSE,CLNUM,RELEAS,RLNUM
 0010               *
 0011               *      PROCEDURE ONE
 0012               *
 0013       0000'   PROC1  EQU $
 0014 0000 2FE0            XOP   @ALUNO,15           ACCESS LUNO NUMBER

 0015 0004 D020            MOVB  @ALNUM,R0           MOVE LUNOO NUMBER TO PRB
      0006 0000
 0016 0008 D800            MOVB  R0,@OLNUM
      000A 0000
 0017 000C D800            MOVB  R0,@W1LNUM
      000E 0000
 0018 0010 D800            MOVB  R0,@W2LNUM
      0012 0000
 0019 0014 D800            MOVB  R0,@CLNUM
      0016 0000
```

```
0020 0018 D800          MOVB RO,@RLNUM
     001A 0000
0021 001C 2FE0          XOP  @OPEN,15           OPEN LUNO
     001E 0000
0022 0020 2FE0          XOP  @WRITE1,15         WRITE TEXT 1
     0022 0000
0023 0024 0460          B    @PROC2             GO TO PROC 2
     0026 0000
0024                    END
NO ERRORS,       NO WARNINGS
```

```
PROC2          SDSMAC 947075 *E     08:43:22 TUESDAY, AUG 02, 19XX
                                                             PAGE 0001
 0001                *
 0002                *      PROCEDURE  SEGMENT 2
 0003                *
 0004                       IDT   'PROC2'
 0005                       DEF   PROC2
 0006                       REF   WRITE2,CLOSE,RELEAS
 0007                *
 0008                *      PROCEDURE TWO
 0009                *
 0010       0000'    PROC2  EQU   $
 0011 0000 2FE0             XOP   @WRITE2,15  WRITE TEXT 2
      0002 0000
 0012 0004 2FE0             XOP   @CLOSE,15   CLOSE LUNO
      0006 0000
 0013                *      XOP   @RELEAS,15  LEASE LUNO**DUMMY IF NOT FILE**
 0014 0008 2FE0             XOP   @EOT,15     END OF TASK
      000A 000C'
 0015                *
 0016                *      CONSTANT DATA
 0017                *
 0018 000C   04      EOT    BYTE 4,0          END OF TASK
      000D   00
 0019                       END
NO ERRORS
```

The following Link Editor listing documents two Link Editor executions: The first Link Editor execution links task segment 1 and procedure segments 1 and 2. The second Link Editor execution links task segment 2 with procedure segments 1 and 2. Thus TASK1 (task segment 1 and procedure segments 1 and 2) shares procedures 1 and 2 with TASK2.

```
TI 990/10 SDSLNK 936060 V2   08/02/XX  09:21:27                 PAGE  1
COMMAND LIST

        FORMAT     IMAGE
        PROCEDURE  PROC1
        INCLUDE    .DEMO.OBJECT.PROC1
        PROCEDURE  PROC2
        INCLUDE    .DEMO.OBJECT.PROC2
        TASK       TASK1
        INCLUDE    .DEMO.OBJECT.TASK1
        END
```

```
TI 990/10 SDSLNK 936060 V2   08/02/XX  09:21:27                    PAGE  2
LINK MAP

CONTROL FILE = .DEMO.LNKCNTRL.TASK1

LINKED OUTPUT FILE = .DEMO.PROG

LIST FILE = .DEMO.LIST.LINK1

NUMBER OF OUTPUT RECORDS = 3

OUTPUT FORMAT = IMAGE


TI 990/10 SDSLNK 936060V2   08/02/XX  09:21:27                     PAGE  3

PROCEDURE 1, PROC1   ORIGIN = 0000  LENGTH = 0024  (PROCEDURE ID = 1)

MODULE     NO   ORIGIN   LENGTH     TYPE       DATE       TIME      CREATOR

PROC1      1     0000     0024     INCLUDE   08/02/XX   08:41:22   SDSMAC
           2     0024     0000     INCLUDE   08/02/XX   08:41:26   SDSMAC

                            D E F I N I T I O N S

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

PROC1   0000   1


TI 990/10 SDSLNK 936060 V2   08/02/XX  09:21:27                    PAGE  4

PROCEDURE 2, PROC2    ORIGIN = 0040  LENGTH = 000E  (PROCEDURE ID = 2)

MODULE     NO   ORIGIN    LENGTH     TYPE       DATE       TIME      CREATOR

PROC2      3     0040      000E    INCLUDE    08/02/XX  08:43:22   SDSMAC
           4     004E      0000    INCLUDE    08/02/XX  08:43:25   SDSMAC


                            D E F I N I T I O N S

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

PROC2   0040   3

TI 990/10 SDSLNK 936060 V2   08/02/XX  09:21:27                      PAGE 5

PHASE 0, TASK1     ORIGIN = 0060  LENGTH = 00B2 (TASK ID = 1)

MODULE     NO  ORIGIN     LENGTH      TYPE      DATE         TIME      CREATOR

TASK1      5    0060       00B2     INCLUDE  08/02/XX     08:37:34    SDSMAC
```

```
                         D E F I N I T I O N S

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

ALNUM     0069   5   *ALUNO   0066   5     CLNUM     00B7   5   CLOSE      00B4   5
OLNUM     008D   5    OPEN    008A   5    *RELEAS    00C2   5   RLNUM      00C5   5
W1LNUM    009B   5    W2LNUM  00A9   5     WRITE1    0098   5   WRITE2     00A6   5


**** LINKING COMPLETED


TI 990/10 SDSLNK 936060 V2   08/02/XX   09:24:27                        PAGE  1
COMMAND LIST

        FORMAT   IMAGE
        PROCEDURE PROC1
        DUMMY
        INCLUDE    .DEMO.OBJECT.PROC1
        PROCEDURE PROC2
        DUMMY
        INCLUDE    .DEMO.OBJECT.PROC2
        TASK       TASK2
        INCLUDE    .DEMO.OBJECT.TASK2
        END

TI 990/10 SDSLNK 936060 V2   08/02/XX   09:24:21                        PAGE  2
LINK MAP

CONTROL FILE = .DEMO.LNKCNTRL.TASK2

LINKED OUTPUT FILE = .DEMO.PROG

LIST FILE = .DEMO.LIST.LINK2

NUMBER OF OUTPUT RECORDS = 1

OUTPUT FORMAT = IMAGE

TI 990/10 SDSLNK 936060 V2   08/02/XX   09:24:21                        PAGE  3

PROCEDURE 1, PROC1     ORIGIN = 0000  LENGTH = 0024, DUMMY   (PROCEDURE ID = 1)

MODULE      NO    ORIGIN     LENGTH      TYPE        DATE        TIME       CREATOR

PROC1       1      0000       0024     INCLUDE     08/02/XX    08:41:22    SDSMAC
            2      0024       0000     INCLUDE     08/02/XX    08:41:26    SDSMAC

                           D E F I N I T I O N S

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

PROC1    0000   1
```

```
TI 990/10 SDSLNK 936060 V2   08/02/XX  09:24:21                      PAGE  4

PROCEDURE 2, PROC2    ORIGIN = 0040  LENGTH = 000E, DUMMY  (PROCEDURE ID = 2)

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE       TIME     CREATOR

PROC2      3      0040      000E     INCLUDE     08/02/XX  08:43:22  SDSMAC
           4      004E      0000     INCLUDE     08/02/XX  08:43:25  SDSMAC

                             D E F I N I T I O N S

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

PROC2  0040  3

TI 990/10 SDSLNK 936060 V2   08/02/XX  09:24:21                    PAGE  5

PHASE 0, TASK2    ORIGIN = 0060  LENGTH = 00B6   (TASK ID = 2)

MODULE     NO  ORIGIN     LENGTH      TYPE       DATE        TIME     CREATOR

TASK2      5    0060        00B5     INCLUDE    08/02/XX    08:40:17  SDSMAC

                             D E F I N I T I O N S

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

ALNUM    0069  5   *ALUNO       0066  5     CLNUM    00B7  5     CLOSE   00B4  5
OLNUM    008D  5    OPEN        008A  5    *RELEAS   00C2  5     RLNUM   00C5  5
W1LNUM   009B  5    W2LNUM      00A9  5     WRITE1   0098  5     WRITE2  00A6  5

**** LINKING COMPLETED
```

The following listing is a map of the program file used in the previous link edit runs.

```
     FILE MAP OF .DEMO.PROG
       TODAY IS 09:27:09 TUESDAY, AUG 02, 19XX

TASKS:
 ID    NAME    LENGTH LOAD   PRI    SPMRD   OVLY  P1/SAME P2/SAME    INSTALL DATE
 01   TASK1     00B2  0060   04     NNNYN         01/Y    02/Y     8/ 2/26    0: 0: 0
 02   TASK2     00B6  0060   04     NNNYN         01/Y    02/Y     8/ 2/26    0: 0: 0
PROCEDURES:
 ID    NAME    LENGTH LOAD         RES  D                            INSTALL DATE
 01   PROC1     0024  0000          N   N                          8/ 2/26    0: 0: 0
 02   PROC2     000E  0040          N   N                          8/ 2/26    0: 0: 0
OVERLAYS:
 ID    NAME    LENGTH LOAD         MAP  D   OVLY                     INSTALL DATE
```

The following listings are sample outputs from TASK2 and TASK1, respectively.

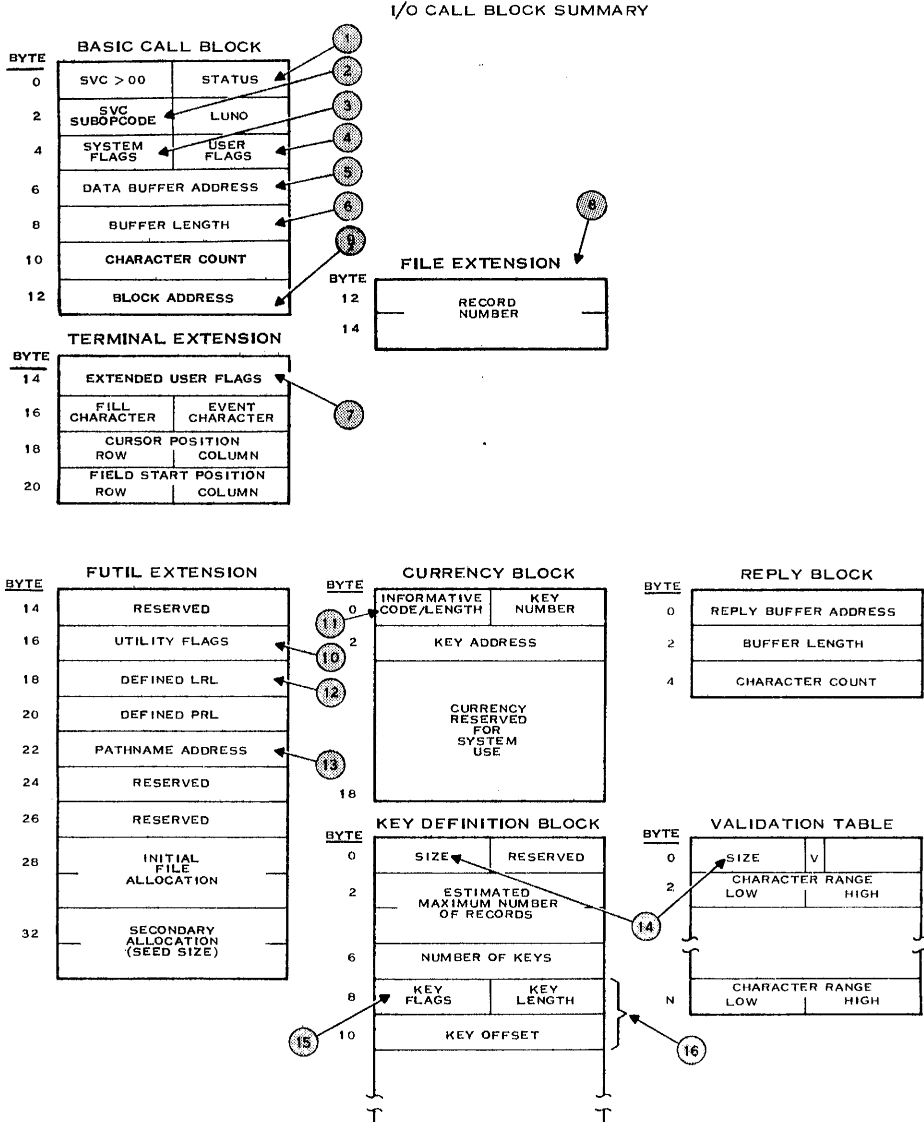
```
I AM TASK TWONOW I AM THROUGH
```

```
I AM TASK ONENOW I AM DONE
```

Appendix E

File and Device I/O SVC Call Blocks

This appendix shows the call blocks required for all I/O SVCs. The SVC opcode for all I/O SVCs is 00, placed in byte 0. The subopcode in byte 2 determines the particular SVC function. Refer to Sections 9 and 10 for detailed information.



Notes:

1. The status codes returned are zero if no errors occurred, or one of the error codes found in Volume VI.

2. The subopcode can have the following values:
   >00 Open (see also notes 12, 15, 30)
   >01 Close (see also note 30)
   >02 Close, write EOF
   >03 Open and rewind (see also notes 12, 15, 30)
   >04 Close and unload
   >05 Read device or file characteristics (see also note 30)
   >06 Forward space (see also note 30)
   >07 Backward space (see also note 30)
   >08 Extended device control (see also note 23)
   >09 Read ASCII (see also note 30)
   >0A Read direct (see also note 30)
   >0B Write ASCII
   >0C Write direct
   >0D Write EOF
   >0E Rewind (see also note 30)
   >0F Unload (see also note 21)
   >10 Rewrite (see also note 18)
   >11 Modify access privileges (see also note 30)
   >12 Open extend (see also note 18)
   >15 Modify device characteristics (see also note 34)
   >4A Unlock record (see also note 18)
   >40 Open KIF (see also notes 12, 15, 17)
   >41 Read greater (see also notes 17, 19)
   >42 Read current or by key (see also notes 17, 19)
   >44 Read greater or equal (see also notes 17, 19)
   >45 Read next (see also note 17)
   >46 Insert (see also note 17)
   >47 Rewrite (see also note 17)
   >48 Read previous (see also note 17)
   >49 Delete current or by key (see also notes 17, 19)
   >4A Unlock current (see also note 17)
   >50 Set currency equal (see also notes 17, 19)
   >51 Set currency greater or equal (see also notes 17, 19)
   >52 Set currency greater (see also notes 17, 19)
   >90 Create file
   >91 Assign LUNO (see also note 20)
   >92 Delete file
   >93 Release LUNO (see also note 20)
   >94 Reserved
   >95 Assign new file name
   >96 Unprotect file
   >97 Write protect file
   >98 Delete protect file
   >99 Verify pathname (see also note 20)

>9A	Add alias
>9B	Delete alias
>9C	Define forced write mode for LUNO

3. The system flags are maintained by the system and indicate the following conditions:

Bit	Meaning When Set
0	Call block busy
1	An error has occurred
2	EOF detected on read
3	Event character terminated read
4–7	Reserved

4. The user flags are set by the program to select optional system functions:

Bit	Meaning When Set
0	Initiate I/O
1	Output with reply (see also notes 20, 32)
2	Reserved
3–4	Access privileges for open on files (see also note 22) 00 — Exclusive write 01 — Exclusive all 10 — Shared (see also note 28) 11 — Read only
5	Immediate open for teleprinter devices Do not replace for open on files Lock on read, unlock on write for I/O to files
6	Extended call block for terminals
7	Allow event characters on opens of terminals Blank adjustment on all I/O

5. The Open call returns the device and file type in the buffer address field as follows:

>0000	Dummy
>0001	Teleprinter
>0002	Line Printer
>0003	ASR 733 Cassette
>0004	Card Reader
>0005	VDT
>0006	Disk or SSSD Diskette
>0007	Communications Device
>0008	Magnetic Tape
>01FF	Sequential File
>02FF	Relative Record File
>03FF	Key Indexed File
>04FF	Directory File
>05FF	Program File
>06FF	Image File

6. The Open call returns the length of record that can be handled by device or file in this field. It is zero when there is no record length (dummy device) or the record size is unknown (magnetic tape). For files, it is the logical record size defined when the file was created.

7. The extended user flags are used with terminal I/O. The extended call block user flag must be set for these flags to be implemented.

Bit	Meaning When Set
0	Use field start cursor position (see also note 24)
1	High intensity (see also note 24)
2	Blink cursor (see also note 24)
3	Display graphics (see also note 24)
4	8-bit data, carriage control (see also note 25, 29)
5	Return on task edit keys (see also note 26, 29)
6	Beep (see also note 24, 29)
7	Right boundary (see also note 27)
8	Use cursor position (see also note 27)
9	Use fill character (see also note 27)
10	Do not initialize field (see also note 27)
11	Return only on termination character (see also note 27, 29)
12	Do not echo (see also note 27, 29)
13	Character validation (see also note 27, 29, 33)
14	Field begins in error (see also note 27, 29)
15	Warning beep (see also note 27, 29)

8. File extension contains a 2-word record number for relative record file read, write, write EOF; also after open extend.

9. Block address contains the address of:

   a. A reply block if bit 1 of the user flags is set

   b. A validation table if bit 13 of extended flags is set

   c. A currency block address if LUNO is assigned to a KIF

   d. A key definition block if creating a KIF

10. The utility flags are used in assign LUNO and create file operations:

Bit	Meaning When Set
0	File was created by assign with autocreate
1–2	Special usage for create relative record file
	00 — No special usage
	01 — Directory file
	10 — Program file
	11 — Image file
3–4	Scope of LUNO assignment (see also note 20)
	00 — Task local
	01 — Station local
	10 — Global
	11 — Reserved
5	Generate LUNO (see also note 20)
6	Autocreate file on assign LUNO (see also note 35)
7	Reserved
8	Logical record is defined in bytes 18–19
9	Temporary file
10	Forced write file
11–12	Data format for sequential files
	00 — Binary, no blank compression
	01 — Blank suppressed
	10 — Reserved
	11 — Reserved
13	File is expandable
14–15	File type
	00 — Reserved
	01 — Sequential file
	10 — Relative record file
	11 — Key indexed file

11. On return from a KIF operation, it is the informative code. On calling a KIF operation, opcodes >41, >44, >50, >51, >52 require the length of the key (possibly partial) used for the search here.

12. Used if bit 8 of the utility flags is set. Otherwise, bytes 8–9 are used for the logical record length.

13. The pathname buffer has the length in bytes of the pathname (not including the length byte) in the first byte of the buffer. It does not have to be an even word address.

14. The size is the total size of the block or table in bytes, including the size byte.

15. Key flags, as follows:

Bit	Meaning When Set
0-4	Reserved. Set to zero.
5	Key is modifiable.
6	Reserved. Set to zero.
7	Duplicates are allowed.

16. These two words are repeated for each extra key to be defined.
17. KIF only.
18. Sequential and relative record only.
19. On call, informative code byte contains key length.
20. Used for devices.
21. Not used for files.
22. Not used for devices.
23. Special control codes for devices not following standard call block format.
24. All read/write calls.
25. Direct read/write calls.
26. ASCII read only.
27. Read only.
28. Shared access to a sequential file allows read and rewrite, but not write.
29. Used for teleprinters.
30. May be used with KIF.
31. Set bit 1 of user flags to indicate by key.
32. Used to indicate operations by key.
33. Bit 1 of user flags must be set.
34. Buffer address points to the flag words.
35. When the autocreate bit is set, all create file fields must be valid.

Appendix F

SVC Codes

The following list shows all SVCs and their codes. The I/O operations (SVC opcode 00) are identified by a subopcode, shown in Appendix E.

Call	Hexadecimal Code
I/O-Related Calls	
Perform I/O Operation	00
Wait for I/O	01
Wait for Multiple Initiate I/Os	36
Get Event Character	30
Abort I/O	0F
Get Event Character	39, 30
File Utility Request	15
Program Control Calls	
Execute Task	2B
Activate Suspended Task	07
Scheduled Bid Task	1F
Load an Overlay	14
End of Task	04
End of Program	16
Time Delay	02
Change Priority	11
Poll Status of Task In Terminal Task Set	35
Unconditional Wait	06
Activate Time Delay Task	0E
Get Parameters	17
Self-Identification	2E
End Action Status	2F
Reset End Action	3E
Map Program Name to ID	31
Bid Task	05
Do Not Suspend	09

Call	Hexadecimal Code
Memory Control Calls	
Get Memory	12
Release Memory	13
Get Common Data Address	10
Return Common Data Address	1B
Miscellaneous Calls	
Getdata	1D
Putdata	1C
Date and Time	03
System Log	21
Retrieve System Information	3F
Data Conversion Calls	
Convert Binary to Decimal	0A
Convert Decimal to Binary	0B
Convert Binary to Hexadecimal	0C
Convert Hexadecimal to Binary	0D
Privileged Supervisor Calls	
Install Task	25
Install Procedure	26
Install Overlay	27
Delete Task	28
Delete Procedure	29
Delete Overlay	2A
Kill Task	33
Suspend Awaiting Queue Input	24
Read/Write TSB	2C
Read/Write Task	2D
Get System Pointer Table Address	32
Initialize Data and Time	3B
Disk Manager	22
Assign Space on Program File	37
Initialize New Volume	38
Install Disk Volume	20
Unload Disk Volume	34
Direct Disk I/O	00

Alphabetical Index

Introduction

HOW TO USE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — Reference to Sections of the manual appear as "Sections x" with the symbol x representing any numeric quantity.

- Appendixes — Reference to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

- Paragraphs — Reference to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.

- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

  Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

  Fx-yy

- Other entries in the Index — References to other entries in the index preceded by the word "See" followed by the referenced entry.

AB Command 11.3.3.1
Abort I/O on LUNO SVC >0F 9.7.5
Access:
  Event Key 9.6.2
  Privileges, File 3.3.4.3, T10-1
  Subroutine, TLC 6.3.8.4
ACNM 6.3.4, 6.3.4.1
Activate Suspended Task SVC >07 . . . . 8.2.4
Activate Task (AT) Command 11.3.4.1
Activate Time Delay Task SVC >0E 8.2.6
Add Alias SVC >00, Subopcode >9A 10.3.10
Add 32-Bit Integers (S$IADD)
  Subroutine 6.3.8.3
Address-Dependent Procedure 2.5.1
Address-Independent Procedure 2.5.1
Arithmetic Utility Subroutine 6.3.8.3
ASB Command 11.3.6.1
Assemble Task 5.3.2
Assembly Language Program
  Segments 5.2
Assign Breakpoints (AB) Command . . 11.3.3.1
Assign LUNO SVC >00,
  Subopcode >91 . . . . 9.5.16, 10.3.2
Assign Simulated Breakpoint (ASB)
  Command 11.3.6.1
AT Command 11.3.4.1
Attach Procedure 5.2.2
Autocreate File SVC >91 . . . . 10.3.2

Background 5.3.5, 6.3.7.2
  Deep 5.3.5
Backward Space Device SVC >00,
  Subopcode >07 . . . . 9.5.8
Backward Space SVC >00,
  Subopcode >07 . . . . 10.2.2.8, 10.2.5.6
BATCH Command 6.4
Batch Command Format 6.4
Batch Stream 6.2.1.3, 6.4
  Example 6.4
  Listing 6.5.3
  Processing 6.4
Bid a Task (S$BIDT) Subroutine 6.3.8.2
Binary-to-Decimal SVC >0A 8.5.1
Binary-to-Hexadecimal SVC >0C 8.5.3
Business System Terminal
  Keyboard Layout FA-5

Call Block:
  Device I/O SVC 9.4
  File I/O SVC 10.2.1
  KIF I/O SVC 10.2.4
  SVC, General 7.2.2
Call, Supervisor . . . . See SVC
Card Reader Character Set TB-9
Change Priority SVC >11 . . . . 8.2.7
Change Write Mode File SVC >9C . . . 10.3.12
Character Code:
  Transformations, Key B.2
  Validation 9.6.6
Character Set:
  Card Reader TB-9
  Display Terminal Graphics TB-7
  Line Printer TB-10
  733/743 Data Terminal TB-8
Close Device EOF SVC >00,
  Subopcode >02 . . . . 9.5.3
Close Device SVC >00,
  Subopcode >01 . . . . 9.5.2
Close EOF SVC >00,
  Subopcode >02 . . . . 10.2.2.3
Close SVC >00,
  Subopcode >01 . . . . 10.2.2.2, 10.2.5.2
Close the Terminal Local File
  (S$CLOS) Subroutine 6.3.8.4
Close Unload Device SVC >00,
  Subopcode >04 . . . . 9.5.5
Close Unload SVC >00,
  Subopcode >04 . . . . 10.2.2.5
Code:
  Country 12.2
  Eight-Bit 12.3
  Informative 10.2.5.24
  Transformations, Key Character B.2
Codes:
  SVC . . . . Appendix F
  Task State TC-1
Coding, SVC 7.2.1
Collating Sequences, KIF 12.5
Command . . . . See name or
  mnemonic of command
  Format, Batch 6.4
  Format, SCI 6.3.1
  Library, SCI 6.1.1
  Privilege, SCI 6.3.7.1
Command Processor:
  Example 6.5.2
  SCI 6.1.3, 6.2.2
Compare Two Strings (S$SCOM)
  Subroutine 6.3.8.1
Convert ASCII to Binary Integer
  (S$INT) Subroutine 6.3.8.1
Convert Binary Integer to ASCII
  (S$IASC) Subroutine 6.3.8.1
Copy String (S$SCPY) Subroutine 6.3.8.1
Country Code 12.2
Create File SVC >00,
  Subopcode >90 . . . . 10.3.1
Currency Block 10.2.4

Data:
  Conversion SVC 8.5
  Keys 9.6.1
  Protection 4.2
  Sharing 2.5.2
Date and Time SVC >03 . . . . 8.6.1
DB Command 11.3.3.2
Debug Mode 11.2
Debugging 11.1
Decimal-to-Binary SVC >0B 8.5.2
Deep Background 5.3.5
Default Menu 6.2.4

Define Write Mode File SVC >00,
  Subopcode >9C ... 10.3.12
Definition, SVC ... 7.2
Delete Alias SVC >00,
  Subopcode >9B ... 10.3.11
Delete and Proceed from Breakpoint
  (DPB) Command ... 11.3.3.3
Delete Breakpoints (DB) Command ... 11.3.3.2
Delete by Key SVC >00,
  Subopcode >49 ... 10.2.5.18
Delete Current SVC >00,
  Subopcode >49 ... 10.2.5.18
Delete File SVC >00,
  Subopcode >92 ... 10.3.3
Delete Protect File SVC >00,
  Subopcode >98 ... 10.3.8
Delete Simulated Breakpoints (DSB)
  Command ... 11.3.6.2
Device:
  Access Logic ... 3.5.2
  File-Oriented ... 3.5.2.2
  Record-Oriented ... 3.5.2.1
Device Dependent I/O ... 9.2
Device Independent I/O ... 9.1
Device I/O:
  Call Block ... 9.4
  SVC ... 9.5
Directory:
  Procedure ... 6.3.7.16
  SCI Procedure ... 6.2.1
Dirty Code ... 2.5.1.3
Disk-Resident:
  Procedure ... 2.4.2
  Task ... 2.4.2
Display Terminal Graphics Character:
  Keyboard Positions:
    911 VDT ... FB-1
    931 VDT ... FB-2
    940 EVT ... FB-3
  Set ... TB-7
Divide 32-Bit Integers (S$IDIV)
  Subroutine ... 6.3.8.3
Do Not Suspend SVC >09 ... 8.2.5
DPB Command ... 11.3.3.3
DSB Command ... 11.3.6.2
DSEG Directive ... 2.5.1.2
DSKMGR ... 3.4
DSR Edit Flags ... 9.9

EBATCH Command ... 6.4
Edit Flag Words ... 9.9
Edit Key:
  System ... 9.6.4
  Task ... 9.6.3
Eight-Bit Code ... 12.3
End Action ... 5.2.4
End Action Status SVC >2F ... 8.2.14
End of Task SVC >04 ... 8.2.2
End Program SVC >16 ... 8.2.9
ERRC ... 7.2.3
Error Code, SVC ... 7.2.3
Error Messages, SCI Procedure ... 6.6
Event Key:
  Access ... 9.6.2
  Flag ... 9.6.2, T9-3
  Option ... 9.6.2
  Programmable ... 9.6.2
Example:
  Command Processor ... 6.5.2
  Programming ... 5.3.6
Execute:
  Program ... 5.3.5
  Task ... 5.3.5
Execute and Halt Task (XHT)
  Command ... 11.3.4.5
Execute in Debug Mode (XD)
  Command ... 11.3.4.4
Execute Task SVC >2B ... 8.2.12
Extended Operation (XOP) ... 7.2

FB Command ... 11.3.5.1
Field Prompt ... 6.1.5, 6.3.4
File:
  Access Privileges ... 3.3.4.3, T10-1
  Keys, Key Indexed ... 10.2.3.1
  Program ... 2.5.5
  Records, Key Indexed ... 10.2.3.2
  Subopcodes, Key Indexed ... 10.2.5
File I/O Call Block ... 10.2.1
File I/O SVC >00:
  Autocreate, Subopcode >91 ... 10.3.2
  Change Write Mode,
    Subopcode >9C ... 10.3.12
  Create, Subopcode >90 ... 10.3.1
  Define Write Mode,
    Subopcode >9C ... 10.3.12
  Delete Protect, Subopcode >98 ... 10.3.8
  Delete, Subopcode >92 ... 10.3.3
  Rename, Subopcode >95 ... 10.3.5
  Unprotect, Subopcode >96 ... 10.3.6
  Write Protect, Subopcode >97 ... 10.3.7
Files:
  Key Indexed ... 10.2.3
  Temporary ... 10.4
FILMGR ... 3.4
Find Byte (FB) Command ... 11.3.5.1
Find Word (FW) Command ... 11.3.5.2
Flags, DSR Edit ... 9.9
Foreground ... 5.3.5, 6.3.7.2
Format, Batch Command ... 6.4
Forward Space Device SVC >00,
  Subopcode >06 ... 9.5.7
Forward Space SVC >00,
  Subopcode >06 ... 10.2.2.7, 10.2.5.5
Function Keys ... 9.6.2
FUTIL ... 3.4
FW Command ... 11.3.5.2

Generic Keycap Names ... Appendix A, TA-1
Get Common Data Address SVC >10 ... 8.3.1
Get Event Key by ID SVC >30 ... 9.7.2
Get Event Key by LUNO SVC >39 ... 9.7.4

Get Memory SVC >12 . . . 8.3.2
Get Parameters SVC >17 . . . 8.2.10
Get Terminal Communications
  Area (S$GTCA) Subroutine . . . 6.3.8.2
Get the I-th Parameter (S$PARM)
  Subroutine . . . 6.3.8.2
Get the Status of the Terminal
  (S$STAT) Subroutine . . . 6.3.8.2
Getdata SVC >1D . . . 8.4.2
Graphics Character:
  Keyboard Positions:
    911 VDT . . . FB-1
    931 VDT . . . FB-2
    940 EVT . . . FB-3
  Set, Display Terminal . . . TB-7

Halt Task (HT) Command . . . 11.3.4.2
Hard Break . . . 9.6.7
Hexadecimal-to-Binary SVC >0D . . . 8.5.4
HT Command . . . 11.3.4.2

Implementation, SVC . . . 7.2
Informative Code . . . 10.2.5.24
Initialize the System Data Base
  (S$NEW) Subroutine . . . 6.3.8.2
Insert SVC >00, Subopcode >46 . . . 10.2.5.15
Install Task . . . 5.3.4
INT . . . 6.3.4.2
International Characters . . . T12-1
International Print File (IPF) Command . . 12.7
Interrupt, Terminal . . . 9.6.7
Intertask Communication SVC . . . 8.4
I/O:
  Device Dependent . . . 9.2
  Device Independent . . . 9.1
  SVC:
    Call Block, Device . . . 9.4
    Call Block, File . . . 10.2.1
    Call Block, KIF . . . 10.2.4
    Device . . . 9.5
    File . . . 10.1
IPF Command . . . 12.7

JISCII . . . 12.3

Key:
  Character Code Transformations . . . B.2
  Event . . . 9.6.2
  System Edit . . . 9.6.4
  Task Edit . . . 9.6.3
Key Indexed File . . . 10.2.3
  Keys . . . 10.2.3.1
  Records . . . 10.2.3.2
  Subopcodes . . . 10.2.5
Key Sequences . . . TA-2
Keyboard Key Types . . . 9.6
  Data . . . 9.6.1
  Event . . . 9.6.2
  Function . . . 9.6.2
  System Edit . . . 9.6.4
  Task Edit . . . 9.6.3
Keyboard Layout:
  Business System Terminal . . . FA-5
  820 KSR . . . FA-6
  911 VDT . . . FA-1
  915 VDT . . . FA-2
  931 VDT . . . FA-4
  940 EVT . . . FA-3
Keyboard Positions, Graphics Character:
  911 VDT . . . FB-1
  931 VDT . . . FB-2
  940 EVT . . . FB-3
Keycap Name Equivalents, 911 VDT . . . TA-3
Keycap Names, Generic . . . Appendix A, TA-1
Keyword, SCI . . . 6.1.5, 6.3.6
KIF Collating Sequences . . . 12.5
KIF, Sequential Placement . . . 10.2.3.4

Language:
  SCI . . . 6.1, 6.1.1
  Variables, SCI . . . 6.3.3
LB Command . . . 11.3.1.1
Library, SCI Procedure . . . 6.2.1
Line Printer Character Set . . . TB-10
Link Task . . . 5.3.3
List Breakpoints (LB) Command . . . 11.3.1.1
List Logical Record (LLR)
  Command . . . 11.3.1.2
List Memory (LM) Command . . . 11.3.1.3
List Simulated Breakpoints (LSB)
  Command . . . 11.3.6.3
List System Memory (LSM)
  Command . . . 11.3.1.4
Listing, Batch Stream . . . 6.5.3
LLR Command . . . 11.3.1.2
LM Command . . . 11.3.1.3
LSB Command . . . 11.3.6.3
LSM Command . . . 11.3.1.4
Load Overlay SVC >14 . . . 8.2.8
Logical Record . . . 3.4.4
Logical Unit Number (LUNO) . . . 3.5
Log-Off Processing . . . 6.2.4
Log-On Processing . . . 6.2.4
Lowercase Mapping . . . 6.3.7.9
LP$x . . . 3.5.3
LUNO . . . 3.5
  SVC >00:
    Assign, Subopcode >91 . . . 10.3.2
    Release, Subopcode >93 . . . 10.3.4

Macro Assembler . . . 5.1
MAD Command . . . 11.3.2.1
MADU Command . . . 11.3.2.2
Map Program Name to ID SVC >31 . . . 8.2.15
Map Synonym (Get Its Value)
  (S$MAPS) Subroutine . . . 6.3.8.2
Mapping, Program . . . 2.2.2
Memory Allocation . . . 2.4.2
Memory Control SVC . . . 8.3
Memory Management . . . 2.4.2
Memory Resident . . . 2.5.5

  - Procedure . . . 2.4.2
  - Task . . . 2.4.2
- Menu . . . 6.2.3, 6.3.7.16
  - Default . . . 6.2.4
  - Procedure . . . 6.2.3
  - Top-Level . . . 6.2.3
- MIR Command . . . 11.3.2.3
- MM Command . . . 11.3.2.4
- Mode, Debug . . . 11.2
- Modify Absolute Disk (MAD) Command . . . 11.3.2.1
- Modify Access Privileges SVC >00, Subopcode >11 . . . 10.2.2.17
- Modify Allocatable Disk Unit (MADU) Command . . . 11.3.2.2
- Modify Device Characteristics SVC >00, Subopcode >15 . . . 9.9
- Modify Internal Registers (MIR) Command . . . 11.3.2.3
- Modify Memory (MM) Command . . . 11.3.2.4
- Modify Program Image (MPI) Command . . . 11.3.2.5
- Modify Relative to File (MRF) Command . . . 11.3.2.6
- Modify System Memory (MSM) Command . . . 11.3.2.7
- Modify Workspace Registers (MWR) Command . . . 11.3.2.8
- MPI Command . . . 11.3.2.5
- MRF Command . . . 11.3.2.6
- MSM Command . . . 11.3.2.7
- Multiple Initiate I/O Wait SVC >36 . . . 9.7.3
- Multiply 32-Bit Integers (S$IMUL) Subroutine . . . 6.3.8.3
- MWR Command . . . 11.3.2.8

- Near Equality Algorithm . . . 6.3.4.6
- News File, SCI . . . 6.2.5
- Nonreentrant . . . 2.5.1.3
- Nonreplicatable . . . 2.5.5

- Opcodes . . . See SVC Subopcodes
- Open Device SVC >00, Subopcode >00 . . . 9.5.1
- Open Extend SVC >00, Subopcode >12 . . . 10.2.2.18
- Open File (S$OPEN) Subroutine . . . 6.3.8.4
- Open Random SVC >00, Subopcode >40 . . . 10.2.5.10
- Open Rewind Device SVC >00, Subopcode >03 . . . 9.5.4
- Open Rewind SVC >00, Subopcode >03 . . . 10.2.2.4, 10.2.5.3
- Open SVC >00, Subopcode >00 . . . 10.2.2.1, 10.2.5.1
- Operation, SCI . . . 6.4
- Overlay . . . 2.2.2.6, 2.5.6, 5.2, 5.2.7
  - Segment . . . 5.2.1

- Pass-Thru Mode . . . 9.8
- PB Command . . . 11.3.3.4
- Physical Record . . . 3.4.5
- Poll Status of Task SVC >35 . . . 8.2.16
- Primitive . . . 6.1.5
  - SCI . . . 6.3.7
  - .BID . . . 6.3.7.2
  - .DATA . . . 6.3.7.3
  - .DBID . . . 6.3.7.2
  - .ELSE . . . 6.3.7.6
  - .ENDIF . . . 6.3.7.6
  - .EOD . . . 6.3.7.3
  - .EOP . . . 6.3.7.1
  - .EVAL . . . 6.3.7.4
  - .EXIT . . . 6.3.7.5
  - .IF . . . 6.3.7.6
  - .LOOP . . . 6.3.7.7
  - .MENU . . . 6.3.7.8
  - .OPTION . . . 6.3.7.9
  - .OVLY . . . 6.3.7.10
  - .PROC . . . 6.2.1.2, 6.3.7.1
  - .PROMPT . . . 6.3.7.11
  - .QBID . . . 6.3.7.2
  - .REPEAT . . . 6.3.7.7
  - .SHOW . . . 6.3.7.12
  - .SPLIT . . . 6.3.7.13
  - .STOP . . . 6.3.7.14
  - .SYN . . . 6.3.7.15
  - .TBID . . . 6.3.7.2
  - .UNTIL . . . 6.3.7.7
  - .USE . . . 6.1.1, 6.2.1, 6.3.7.16
  - .WHILE . . . 6.3.7.7
- Printer Character Set, Line . . . TB-10
- Priority . . . 2.4.1.1
  - Task . . . 8.2.7
- Privilege, SCI Command . . . 6.3.7.1
- Privileges, File Access . . . 3.3.4.3, T10-1
- Procedure:
  - Address-Dependent . . . 2.5.1
  - Address-Independent . . . 2.5.1
  - Attach . . . 5.2.2
  - Directory, SCI . . . 6.1.1, 6.1.2, 6.3.7.16
  - Disk-Resident . . . 2.4.2
  - Error Messages, SCI . . . 6.6
  - Library, SCI . . . 6.1.1, 6.2.1, 6.2.1.4
  - Memory-Resident . . . 2.4.2
  - Menu . . . 6.2.3
  - SCI . . . 6.1.1
  - Segment . . . 2.2, 5.2, 5.2.1
  - Shared . . . 2.5.1
- Proceed from Breakpoint (PB) Command . . . 11.3.3.4
- Processor:
  - Interface Subroutine . . . 6.3.8, See also name or mnemonic of subroutine
  - Subroutine . . . 6.1.4
- Program . . . 2.1
  - Execute . . . 5.3.5
  - File . . . 2.5.5
  - Mapping . . . 2.2.2
  - Segmentation . . . 2.2.1, 2.2.2.1
  - Segments, Assembly Language . . . 5.2
- Program Support SVC . . . 8.1

Programmable Event Key ... 9.6.2
Programming Example ... 5.3.6
Prompt ... 6.1.5
  Field ... 6.3.4
PSEG Directive ... 2.5.1.2
Put Terminal Communications
  Area (S$PTCA) Subroutine ... 6.3.8.2
Putdata SVC >1C ... 8.4.1

QD Command ... 11.3.6.4
Quit Debug Mode (QD) Command ... 11.3.6.4

Read ASCII Device SVC >00,
  Subopcode >09 ... 9.5.9
Read ASCII SVC >00,
  Subopcode >09 ... 10.2.2.9, 10.2.5.7
Read by Key SVC >00,
  Subopcode >42 ... 10.2.5.12
Read Current SVC >00,
  Subopcode >42 ... 10.2.5.12
Read Device Status SVC >00,
  Subopcode >05 ... 9.5.6
Read Direct Device SVC >00,
  Subopcode >0A ... 9.5.10
Read Direct SVC >00,
  Subopcode >0A ... 10.2.2.10, 10.2.5.8
Read File Characteristics SVC >00,
  Subopcode >05 ... 10.2.2.6
Read Greater or Equal SVC >00,
  Subopcode >44 ... 10.2.5.13
Read Greater SVC >00,
  Subopcode >41 ... 10.2.5.11
Read Next SVC >00,
  Subopcode >45 ... 10.2.5.14
Read Previous SVC >00,
  Subopcode >48 ... 10.2.5.17
Record:
  Logical ... 3.4.4
  Physical ... 3.4.5
Records, Key Indexed File ... 10.2.3.2
Reentrant ... 2.2, 2.5.1, 5.2.1
Release LUNO SVC >00,
  Subopcode >93 ... 9.5.17, 10.3.4
Release Memory SVC >13 ... 8.3.3
Release Terminal Communications
  Area (S$RTCA) Subroutine ... 6.3.8.2
Rename File SVC >00,
  Subopcode >95 ... 10.3.5
Replicatable ... 2.5.5
Reset End Action SVC >3E ... 8.2.17
Resume Simulated Task (RST)
  Command ... 11.3.6.5
Resume Task (RT) Command ... 11.3.4.3
Retrieve System Information
  SVC >3F ... 8.6.3
Return Common Data Address
  SVC >1B ... 8.3.4
Return to the System Command
  Interpreter (S$STOP) Subroutine ... 6.3.8.2
Rewind Device SVC >00,
  Subopcode >0E ... 9.5.14
Rewind SVC >00,
  Subopcode >0E ... 10.2.2.14, 10.2.5.9
Rewrite KIF SVC >00,
  Subopcode >47 ... 10.2.5.16
Rewrite SVC >00,
  Subopcode >10 ... 10.2.2.16
Roll-In/Roll-Out ... 2.4.2
RST Command ... 11.3.6.5
RT Command ... 11.3.4.3

SAD Command ... 11.3.1.5
SADU Command ... 11.3.1.6
SCC Command ... 12.8
Scheduled Bid Task SVC >1F ... 8.2.11
Scheduling ... 2.4.1
SCI:
  Command Format ... 6.3.1
  Command Library ... 6.1.1
  Command Privilege ... 6.3.7.1
  Command Procedure ... 6.1.1
  Command Processor ... 6.1.3, 6.2.2
  Interface Subroutine ... 6.3.8.2
  Keyword ... 6.3.6
  Language ... 6.1
  News File ... 6.2.5
  Operation ... 6.4
  Primitive ... 6.3.7
  Procedure ... 6.1.1
    Directory ... 6.1.2, 6.2.1, 6.2.1.4
    Error Messages ... 6.6
    Library ... 6.2.1, 6.2.1.4
  Special Characters ... 6.3.2
  Subroutine ... 6.1.4
  Synonyms ... 6.3.5
  Variables ... 6.3.3
Search Name Correspondence
  Table (S$SNCT) Subroutine ... 6.3.8.2
Segment:
  Overlay ... 5.2.1
  Procedure ... 2.2, 5.2, 5.2.1
  Task ... 2.2, 5.2, 5.2.1
Segmentation, Program ... 2.2.1, 2.2.2.1
Segments:
  Assembly Language Program ... 5.2
  Sharable ... 2.2.2
Self-Identification SVC >2E ... 8.2.13
Sequential Placement KIF ... 10.2.3.4
Set Currency Equal SVC >00,
  Subopcode >50 ... 10.2.5.20
Set Currency Greater or Equal
  SVC >00, Subopcode >51 ... 10.2.5.21
Set Currency Greater SVC >00,
  Subopcode >52 ... 10.2.5.22
Set Synonym Value (S$SETS)
  Subroutine ... 6.3.8.2
Sharable Segments ... 2.2.2
Shared Procedure ... 2.5.1
Show Absolute Disk (SAD)
  Command ... 11.3.1.5
Show Allocatable Disk Unit (SADU)
  Command ... 11.3.1.6

Show Country Code (SCC) Command . . . 12.8
Show Internal Registers (SIR)
  Command 11.3.1.7
Show Panel (SP) Command 11.3.1.8
Show Program Image (SPI)
  Command 11.3.1.9
Show Relative to File (SRF)
  Command 11.3.1.10
Show Value (SV) Command 11.3.1.11
Show Workspace Registers (SWR)
  Command 11.3.1.12
Simulate Task (ST) Command 11.3.6.6
SIR Command 11.3.1.7
SP Command 11.3.1.8
Special Characters, SCI 6.3.2
SPI Command 11.3.1.9
Split List into Components (S$SPLT)
  Subroutine 6.3.8.2
SRF Command 11.3.1.10
ST Command 11.3.6.6
String 6.3.4, 6.3.4.4
  Utility Subroutine 6.3.8.1
Subopcode, I/O SVC 9.1, 10.1
Subroutine:
  Arithmetic Utility 6.3.8.3
  Processor Interface . . . . 6.3.8, See also name or mnemonic of subroutine
  SCI 6.1.4
  SCI Interface 6.3.8.2
  String Utility 6.3.8.1
  TLC Access 6.3.8.4
Subroutines, User 5.2.6
Subtract 32-Bit Integers (S$ISUB)
  Subroutine 6.3.8.3
Supervisor Call . . . . See SVC
SV Command 11.3.1.11
SVC . . . . Section 7, 5.1, 5.2.5
  Abort I/O on LUNO, >0F 9.7.5
  Activate Suspended Task, >07 . . . . . . 8.2.4
  Activate Time Delay Task, >0E 8.2.6
  Binary-to-Decimal, >0A 8.5.1
  Binary-to-Hexadecimal, >0C 8.5.3
  Call Block 7.2.2
    File I/O 10.2.1
    KIF I/O 10.2.4
  Change Priority, >11 . . . . 8.2.7
  Codes . . . . Appendix F
  Coding 7.2.1
  Data Conversion 8.5
  Date and Time, >03 . . . . 8.6.1
  Decimal-to-Binary, >0B 8.5.2
  Definition 7.2
  Device I/O 9.5
  Do Not Suspend, >09 . . . . 8.2.5
  End Action Status, >2F 8.2.14
  End of Task, >04 . . . . 8.2.2
  End Program, >16 . . . . 8.2.9
  Error Code 7.2.3
  Execute Task, >2B 8.2.12
  File I/O 10.1
  Get Common Data Address, >10 . . . . 8.3.1
  Get Event Key by ID, >30 . . . . 9.7.2
  Get Event Key by LUNO, >39 . . . . 9.7.4
  Get Memory, >12 . . . . 8.3.2
  Get Parameters, >17 . . . . 8.2.10
  Getdata, >1D 8.4.2
  Hexadecimal-to-Binary, >0D 8.5.4
  Implementation 7.2
  Intertask Communication 8.4
  Load Overlay, >14 . . . . 8.2.8
  Map Program Name to ID, >31 . . . . . 8.2.15
  Memory Control 8.3
  Poll Status of Task, >35 . . . . 8.2.16
  Program Support 8.1
  Putdata, >1D 8.4.1
  Release Memory, >13 . . . . 8.3.3
  Reset End Action, >3E 8.2.17
  Retrieve System Information, >3F . . . 8.6.3
  Return Common Data
    Address, >1B 8.3.4
  Scheduled Bid Task, >1F 8.2.11
  Self-Identification, >2E 8.2.13
  System Information 8.6
  System Log, >21 . . . . 8.6.2
  Time Delay, >02 . . . . 8.2.1
  Unconditional Wait, >06 . . . . 8.2.3
  Wait for I/O, >01 . . . . 9.7.1
SVC Subopcode:
  Add Alias, >9A 10.3.10
  Assign LUNO, >91 . . . . 9.5.16, 10.3.2
  Autocreate File, >91 . . . . 10.3.2
  Backward Space Device, >07 . . . . 9.5.8
  Backward Space, >07 . . . . 10.2.2.8, 10.2.5.6
  Change Write Mode File, >9C 10.3.12
  Close Device EOF, >02 . . . . 9.5.3
  Close Device, >01 . . . . 9.5.2
  Close EOF, >02 . . . . 10.2.2.3
  Close Unload Device, >04 . . . . 9.5.5
  Close Unload, >04 . . . . 10.2.2.5
  Close, >01 . . . . 10.2.2.2, 10.2.5.2
  Create File, >90 . . . . 10.3.1
  Define Write Mode File, >9C 10.3.12
  Delete Alias, >9B 10.3.11
  Delete by Key, >49 . . . . 10.2.5.18
  Delete Current, >49 . . . . 10.2.5.18
  Delete File, >92 . . . . 10.3.3
  Delete Protect File, >98 . . . . 10.3.8
  Forward Space Device, >06 . . . . 9.5.7
  Forward Space, >06 . . . . . 10.2.2.7, 10.2.5.5
  Insert, >46 . . . . 10.2.5.15
  Modify Access Privileges, >11 . . . 10.2.2.17
  Modify Device Characteristics, >15 . . . . 9.9
  Multiple Initiate I/O Wait, >36 . . . . 9.7.3
  Open Device, >00 . . . . 9.5.1
  Open Random, >40 . . . . 10.2.5.10
  Open Rewind Device, >03 . . . . 9.5.4
  Open Rewind, >03 . . . . . . 10.2.2.4, 10.2.5.3
  Open, >00 . . . . 10.2.2.1, 10.2.5.1
  Read ASCII Device, >09 . . . . 9.5.9
  Read ASCII, >09 . . . . 10.2.2.9, 10.2.5.7
  Read by Key, >42 . . . . 10.2.5.12
  Read Current, >42 . . . . 10.2.5.12

Read Device Status, >05 . . . 9.5.6
Read Direct Device, >0A . . . 9.5.10
Read Direct, >0A . . . 10.2.2.10, 10.2.5.8
Read File Characteristics, >05 . . . 10.2.2.6
Read Greater or Equal, >44 . . . 10.2.5.13
Read Greater, >41 . . . 10.2.5.11
Read Next, >45 . . . 10.2.5.14
Read Previous, >48 . . . 10.2.5.17
Release LUNO, >93 . . . 9.5.17, 10.3.4
Rename File, >95 . . . 10.3.5
Rewind Device, >0E . . . 9.5.14
Rewind, >0E . . . 10.2.2.14, 10.2.5.9
Rewrite KIF, >47 . . . 10.2.5.16
Rewrite, >10 . . . 10.2.2.16
Set Currency Equal, >50 . . . 10.2.5.20
Set Currency Greater or
Equal, >51 . . . 10.2.5.21
Set Currency Greater, >52 . . . 10.2.5.22
Unload Device, >0F . . . 9.5.15
Unload, >0F . . . 10.2.2.15
Unlock Current, >4A . . . 10.2.5.19
Unlock, >4A . . . 10.2.2.19
Unprotect File, >96 . . . 10.3.6
Verify Device Name, >99 . . . 9.5.18
Verify Pathname, >99 . . . 10.3.9
Write ASCII to Device, >0B . . . 9.5.11
Write Direct Device, >0C . . . 9.5.12
Write Direct, >0C . . . 10.2.2.12
Write EOF to Device, >0D . . . 9.5.13
Write EOF, >0D . . . 10.2.2.13
Write Protect File, >97 . . . 10.3.7
SWR Command . . . 11.3.1.12
Synchronization, Task . . . 8.2.3
Synonym . . . 6.1.5, 6.3.7.15
$$BC . . . 6.3.7.2
$$CC . . . 6.3.7.2
Synonyms, SCI . . . 6.3.5
System Edit Key . . . 9.6.4
System Information SVC . . . 8.6
System Log SVC >21 . . . 8.6.2
S$BIDT Subroutine . . . 6.3.8.2
S$CLOS Subroutine . . . 6.3.8.4
S$GTCA Subroutine . . . 6.3.8.2
S$IADD Subroutine . . . 6.3.8.3
S$IASC Subroutine . . . 6.3.8.1
S$IDIV Subroutine . . . 6.3.8.3
S$IMUL Subroutine . . . 6.3.8.3
S$INT Subroutine . . . 6.3.8.1
S$ISUB Subroutine . . . 6.3.8.3
S$MAPS Subroutine . . . 6.3.8.2
S$NEW Subroutine . . . 6.3.8.2
S$OPEN Subroutine . . . 6.3.8.4
S$PARM Subroutine . . . 6.3.8.2
S$PTCA Subroutine . . . 6.3.8.2
S$RTCA Subroutine . . . 6.3.8.2
S$SCOM Subroutine . . . 6.3.8.1
S$SCPY Subroutine . . . 6.3.8.1
S$SETS Subroutine . . . 6.3.8.2
S$SNCT Subroutine . . . 6.3.8.2
S$SPLT Subroutine . . . 6.3.8.2
S$STAT Subroutine . . . 6.3.8.2
S$STOP Subroutine . . . 6.3.8.2
S$WEOL Subroutine . . . 6.3.8.4
S$WRIT Subroutine . . . 6.3.8.4

Task . . . 2.1
Assemble . . . 5.3.2
Disk-Resident . . . 2.4.2
Edit Key . . . 9.6.3
Execute . . . 5.3.5
Install . . . 5.3.4
Link . . . 5.3.3
Memory-Resident . . . 2.4.2
Priority . . . 8.2.7
Scheduling . . . 2.4.1
Segment . . . 2.2, 5.2, 5.2.1
Sentry . . . 2.4.1, 2.4.1.2
State Codes . . . TC-1
Synchronization . . . 8.2.3
Temporary Files . . . 10.4
Terminal Interrupt . . . 9.6.7
Time Delay SVC >02 . . . 8.2.1
TLC Access Subroutine . . . 6.3.8.4
Top-Level Menu . . . 6.2.3
Transaction Logging . . . 4.3
Transfer Vector . . . 5.2.3

Unconditional Wait
SVC >06 . . . 8.2.3
Unload Device SVC >00,
Subopcode >0F . . . 9.5.15
Unload SVC >00,
Subopcode >0F . . . 10.2.2.15
Unlock Current SVC >00,
Subopcode >4A . . . 10.2.5.19
Unlock SVC >00,
Subopcode >4A . . . 10.2.2.19
Unprotect File SVC >00,
Subopcode >96 . . . 10.3.6
User Subroutines . . . 5.2.6

Validation, Character Code . . . 9.6.6
Variables, SCI Language . . . 6.3.3
Verify Device Name SVC >00,
Subopcode >99 . . . 9.5.18
Verify Pathname SVC >00,
Subopcode >99 . . . 10.3.9

Wait for I/O SVC >01 . . . 9.7.1
Write ASCII to Device SVC >00,
Subopcode >0B . . . 9.5.11
Write Direct Device SVC >00,
Subopcode >0C . . . 9.5.12
Write Direct SVC >00,
Subopcode >0C . . . 10.2.2.12
Write End-of-Line to the Terminal Local
File (S$WEOL) Subroutine . . . 6.3.8.4
Write EOF SVC >00,
Subopcode >0D . . . 10.2.2.13
Write EOF to Device SVC >00,
Subopcode >0D . . . 9.5.13

Write Protect File SVC >00,
Subopcode >97 . . . . 10.3.7
Write to the Terminal Local
File (S$WRIT) Subroutine 6.3.8.4

XD Command 11.3.4.4
XHT Command 11.3.4.5
XOP Instruction 5.1, 7.2

$$BC Synonym 6.3.7.2
$$CC Synonym 6.3.7.2

.BID Primitive 6.3.7.2
.DATA Primitive 6.3.7.3
.DBID Primitive 6.3.7.2
.ELSE Primitive 6.3.7.6
.ENDIF Primitive 6.3.7.6
.EOD Primitive 6.3.7.3
.EOP Primitive 6.3.7.1
.EVAL Primitive 6.3.7.4
.EXIT Primitive 6.3.7.5
.IF Primitive 6.3.7.6
.LOOP Primitive 6.3.7.7
.MENU Primitive 6.3.7.8
.OPTION Primitive 6.3.7.9
.OVLY Primitive 6.3.7.10
.PROC Primitive 6.2.1.2, 6.3.7.1
.PROMPT Primitive 6.3.7.11
.QBID Primitive 6.3.7.2
.REPEAT Primitive 6.3.7.7
.SHOW Primitive 6.3.7.12
.SPLIT Primitive 6.3.7.13
.STOP Primitive 6.3.7.14
.SYN Primitive 6.3.7.15
.S$NEWS 6.2.5
.TBID Primitive 6.3.7.2
.UNTIL Primitive 6.3.7.7
.USE Primitive 6.1.1, 6.2.1, 6.3.7.16
.WHILE Primitive 6.3.7.7

733/743 Data Terminal Character Set . . . TB-8
820 KSR Keyboard Layout FA-6
911 VDT:
Graphics Character Keyboard
Positions FB-1
Keyboard Layout FA-1
Keycap Name Equivalents TA-3
915 VDT Keyboard Layout FA-2
931 VDT:
Graphics Character Keyboard
Positions FB-2
Keyboard Layout FA-4
940 EVT:
Graphics Character Keyboard
Positions FB-3
Keyboard Layout FA-3

USER'S RESPONSE SHEET

Manual Title: DX10 Operating System Application Programming Guide, Volume III (946250-9703)

Manual Date: January 1985 **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

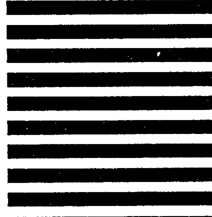
NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

CUT ALONG LINE

TAPE EDGE TO SEAL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP
ATTN: PUBLISHING CENTER
P.O. Box 2909 M/S 2146
Austin, Texas 78769-9990

FOLD