IOS™ Software Internal Reference Manual

SM–0046 G

Cray Research, Inc.

Copyright © 1980, 1981, 1982, 1983, 1984, 1986, 1987, 1988 by Cray Research, Inc. This manual or parts thereof may not be reproduced unless permitted by contract or by written permission of Cray Research, Inc.

CRAY, CRAY-1, SSD, and UNICOS are registered trademarks and CFT, CFT77, CFT2, COS, CRAY-2, CRAY X-MP, CRAY X-MP EA, CRAY Y-MP, CSIM, HSX, IOS, SEGLDR, and SUPERLINK are trademarks of Cray Research, Inc.

HYPERchannel and NSC are registered trademarks of Network Systems Corporation. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark of AT&T.

The UNICOS operating system is derived from the AT&T UNIX System V operating system. UNICOS is also based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120

NEW FEATURES

Release 4.2 of the I/O Subsystem (IOS) includes several enhancements of and additions to previous versions of the subsystem. Those enhancements that affect the I/O Subsystem Internal Reference Manual are presented in this description.

Drivers have been added to the IOS to support the HSX High-speed External Communications channel and the VMEbus. The HSX driver supports the CRI HSX channel. The VMEbus driver allows a VMEbus-based front-end processor connected to a CRI VMEbus interface to communicate with a Cray computer system. Sections 12 and 13 have been added to document this support.

The NSC HYPERchannel driver links a Cray mainframe and a front-end through the NSC HYPERchannel. The driver allows multiple front-end computers to be connected to one physical MIOP channel pair. The FEI driver provides an FEI connection for UNICOS. This connection parallels the NSC logical path connection. The driver allows front-end stations to communicate with the UNICOS Station Call Processor (USCP) by using the SCP protocol. To clarify the special features of these drivers, the NSC HYPERchannel driver and the Front-end Interface (FEI) logical path driver are each documented in a separate section for release 4.2.

The Tape Exec (TEX) software to process tape I/O requests from the mainframe and the block multiplexer (BMX) channel interface software have been restructured for the 4.2 release. The documentation for TEX and the BMX driver has also been extensively updated and restructured.

IOS release 4.2 supports the RD-10 and DD-40 disk storage units. All information regarding disk I/O has been revised to document this support.

RECORD OF REVISION

CRAY RESEARCH, INC.

PUBLICATION NUMBER SM-0046

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
1345 Northland Drive
Mendota Heights, Minnesota 55120

Revision	Description
	November 1980 - Original printing.
A	June 1981 - This rewrite incorporates the interactive station, the division of debugger code into two decks, the PATCH and LISTO commands, descriptions of the main disk overlays, the Concentrator Table, the dynamic allocation of overlay space in Local Memory, the FLUSH service function, and other miscellaneous technical and editorial changes to bring this manual into agreement with the version 1.10 IOS software. This manual obsoletes all previous printings.
B	June 1982 - This reprint incorporates IOS tape support software, especially the interface to block multiplexer channels, the Tape Exec software, and their respective tables. Other new features include on-line diagnostics; error channel processing; the OUTCALL and ASLEEP Kernel service functions; station message support, the LOAD, ISFIELD, FLDADD, and FLDSUB macros; the LISTP and DKDMP analyst aids; and miscellaneous technical and editorial changes to bring this manual into agreement with the version 1.11 IOS software. This manual obsoletes all previous printings.
C	May 1983 - This reprint with revision supports APML loader and Tape Exec updates, error recovery enhancements, the addition of deadstart from 80 Mbyte disk on IOS, Local Memory refresh, station debug commands, the A130OI Kernel service function, new concentrator software (NSC), startup channel and device configuration changes, and the capability of accepting bad data from disk. It incorporates an entirely new Block Mux section and associated tables, new history trace information and format, a new section on SYSDUMP, and a section provided for dump analysis. This manual obsoletes all previous printings.

C-01 February 1984 - This change packet brings the manual into agreement with version 1.13 of COS and supports disk striping, multitasking, the ECHCP diagnostic, changes to the MGET and MPUT macros, dataset disposition to Peripheral Expander disk, recursive error recovery for on-line tape, and an IOS on-line mainframe channel test. This change packet also includes miscellaneous technical and editorial changes.

D December 1984 - This reprint with revision brings the manual into agreement with version 1.14 of COS and supports the addition of DD-49 disk controlling software, tape end-read functions, trace event codes and parameters, the deck OVLNUM, and the D4STIO, D4SEEK, STATIO, and SEND functions. An entirely new section is included on User Channel I/O. Changes are incorporated for STAGEIN and STAGEOUT tasks, concentrator software, the structure of the interactive concentrator, the FIELD, TABLE, and REGDEFS macros, Kernel Disk I/O, the MGET and MPUT functions, and the I/O processor intercommunication function codes. Changes and additions have been made to the device request stream and field engineering diagnostics. This revision also contains documentation of the Integrated Support Processor (ISP). The completed ISP code will not be available until a later date, when you will be notified in a letter accompanying the code. ISP manuals will be available when the completed ISP code is released. This manual obsoletes all previous printings.

E January 1986 - This reprint with revision brings the manual into agreement with COS version 1.15 and obsoletes all previous printings. Information has been added to support: DD-39 disk control and error recovery; the new TRANSFER Kernel service function; the CLEAR, COPY, and RETREG macros; the new BYPIO trace parameter; and the NSCNCIO overlay in the NSC activity of the front-end concentrator. Changes have been made to support changes to: the AWAKE, CALL, CREATE, GETMEM, HSPR, HSPW, and PUSH Kernel service functions; some trace event parameters; Tape Exec to support cartridge-type tape drives; the IOS station global symbols; and the FIELD macro. Information has been deleted about: the D4SEEK, D4STIO, STATIO, and SYNC kernel service functions; and the F80M diagnostic. Appendix B was deleted and relevant information moved into the body of the manual. To support the Cray operating system UNICOS, changes to SYSDUMP were made. Many miscellaneous technical and editorial changes have also been made.

F April 1987 - This revision brings the manual into agreement with COS version 1.16 and UNICOS version 2.0. Information has been added on Target Memory, DD-19 and DD-29 On-line Diagnostics, and Trace Event Codes supporting concentrator status. The CONCIO Activity Description and the NSC overlay subsections in section 7 have been rewritten and FEI Logical Path Activity description has been added; the Field Engineering Diagnostic information has been removed from appendix B; and changes have been made to the DKDMP overlay in section 11. All trademarks are now documented in the record of revision. Many miscellaneous technical and editorial changes have also been made.

G September 1988 - This reprint with revision brings the manual into agreement with IOS 4.2. Sections 4 and 5 were extensively rewritten to support the restructure of the tape and BMX software. The information on NSC HYPERchannel and Front-end Interface logical path activity (previously part of section 7) were put into sections by themselves (sections 10 and 11, respectively). Section 12 on the HSX channel interface and section 13 on the VMEbus driver were both newly added. Appendix B has been renamed "IOS Confidence Utilities". Examples have been added to sections 2 and 14. Many miscellaneous technical and editorial changes have also been made.

PREFACE

This manual describes the software executing in the Cray I/O Subsystem (IOS). This software can be divided into the following general categories:

- The Kernel
- Disk I/O software
- Tape I/O software
- Block multiplexer channel
- The IOS station
- The front-end concentrator
- User channel I/O software
- Drivers
- Communications channel software

In addition to the preceding categories, this manual also contains sections describing the interactive station, the program library and macros, and debugging tools for working with the software.

Cray Research, Inc. (CRI) publications that provide additional information on the IOS are as follow:

Publication	Manual Title
SM-0007	IOS Table Descriptions Internal Reference Manual
SG-0051	I/O Subsystem (IOS) Operator's Guide for COS
SG-2005	I/O Subsystem (IOS) Operator's Guide for UNICOS
SM-0042	Cray Front-end Protocol Internal Reference Manual
SM-0043	COS Operational Procedures Reference Manual
SM-0044	Operational Aids Reference Manual
SG-2018	UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems
HR-0030	I/O Subsystem Model B Hardware Reference Manual
HR-0077	Disk Systems Hardware Reference Manual
HR-0081	I/O Subsystem Model C Hardware Reference Manual

Supplemental information on the IOS is available in the IOS hardware reference manual for your site. This manual also assumes you are familiar with and experienced in coding APML as described in the APML Assembler Reference Manual, CRI publication SM-0036. All publications referenced in this manual are CRI publications unless otherwise noted.

The following IBM form numbers are helpful in understanding the capabilities of the Block Multiplexer channel:

Form Number	Manual Title
GA22-6974-4	IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers'
GA22-6974-5	Information (IBM OEMI Channel Standard)
GA22-7000-5	IBM System/370 Principles of Operation

READER COMMENTS

If you have any comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Call our Technical Publications department at (612) 681-5729 during normal business hours (Central Time).
- Send us electronic mail from a UNICOS or UNIX system, using one of the following electronic mail addresses:

  ihnp4!cray!publications or **sun!tundra!hall!publications**

- Use the postage-paid Reader Comment form at the back of this manual.
- Write to us at the following address:

  Cray Research, Inc.
  Technical Publications Department
  1345 Northland Drive
  Mendota Heights, Minnesota 55120

We value your comments and will respond to them promptly.

CONTENTS

PREFACE vii

1. INTRODUCTION 1-1

1.1 HARDWARE SPECIFICATIONS 1-3
1.2 SYSTEM CONFIGURATION 1-4
1.3 REGISTER ASSIGNMENTS 1-6
1.4 TERMINOLOGY 1-6
1.5 FORMAL SYNTAX CONVENTIONS 1-8

2. THE KERNEL 2-1

2.1 LOCAL MEMORY USAGE 2-1
2.1.1 Local Memory scrubbing 2-2
2.2 BUFFER MEMORY USAGE 2-3
2.2.1 System Directory 2-3
2.2.2 Message areas 2-5
2.2.3 Kernel area 2-6
2.2.4 Buffer Memory resident datasets 2-6
2.3 TARGET MEMORY 2-6
2.4 ACTIVITY-SOFTWARE STACKING 2-7
2.5 DEMON ACTIVITIES 2-8
2.6 OVERLAYS 2-9
2.7 INTERRUPT PROCESSING 2-11
2.8 IOP CENTRAL PROCESSOR QUEUING AND ACTIVITY DISPATCHING . 2-12
2.9 KERNEL SERVICE REQUESTS 2-12
2.9.1 General service functions 2-12
2.9.2 Memory allocation and deallocation 2-12
2.9.3 I/O operations 2-13
2.9.4 Function descriptions 2-16
2.9.4.1 ALERT function (15) 2-16
2.9.4.2 ASLEEP function (14) 2-18
2.9.4.3 AWAKE function (16) 2-19
2.9.4.4 A130OI function (24) 2-21
2.9.4.5 BGET function (32) 2-22
2.9.4.6 BRET function (33) 2-23
2.9.4.7 CALL function (50) 2-23
2.9.4.8 CREATE function (55) 2-24
2.9.4.9 FIND function (53) 2-26
2.9.4.10 FLUSH function (54) 2-26
2.9.4.11 GETDAL function (26) 2-27

2.9.4 Function descriptions (continued)
2.9.4.12 GETMEM function (30) 2-27
2.9.4.13 GIVEUP function (4) 2-28
2.9.4.14 GOTO function (51) 2-29
2.9.4.15 HSPR function (42) 2-30
2.9.4.16 HSPW function (43) 2-32
2.9.4.17 MGET function (35) 2-33
2.9.4.18 MOSR function (46) 2-34
2.9.4.19 MOSW function (47) 2-35
2.9.4.20 MPUT function (36) 2-36
2.9.4.21 MSG function (20) 2-37
2.9.4.22 MSGR function (21) 2-37
2.9.4.23 OUTCALL function (37) 2-38
2.9.4.24 OUTPUT function (22) 2-39
2.9.4.25 PAUSE function (7) 2-40
2.9.4.26 POLL function (44) 2-40
2.9.4.27 POP function (2) 2-41
2.9.4.28 PUSH function (1). 2-42
2.9.4.29 RECEIVE function (25) 2-43
2.9.4.30 RELDAL function (27) 2-43
2.9.4.31 RELMEM function (31) 2-44
2.9.4.32 RESPOND function (17) 2-44
2.9.4.33 RETURN function (52) 2-45
2.9.4.34 SEND function (34) 2-45
2.9.4.35 TERM function (3) 2-46
2.9.4.36 TPUSH function (11) 2-46
2.9.4.37 TRANSFER function (45) 2-47
2.10 CLOCK FUNCTIONS 2-49
2.10.1 Real-time clock interrupt handler 2-49
2.10.2 Clock demon 2-49
2.10.3 System event timer 2-50
2.11 IOP DEADSTART 2-51
2.12 STATISTICS 2-52
2.13 COMMUNICATION AMONG IOPs 2-52
2.14 MIOP-MAINFRAME COMMUNICATION CHANNEL 2-54
2.14.1 MIOP-mainframe communication initialization . . 2-55
2.14.2 Input channel from the mainframe 2-55
2.14.3 Input packet disposition 2-56
2.14.4 Output channel to the mainframe 2-56
2.15 ERROR PROCESSING 2-57
2.15.1 Error channel processing (IOS Serial No. 21 and below) 2-57
2.15.1.1 Interrupt answering 2-58
2.15.1.2 Retrieving error log information . . . 2-59
2.15.2 Error logging (IOS Serial No. 21 and up) 2-59

3. DISK INPUT/OUTPUT 3-1

3.1 REQUEST PROCESS OVERVIEW 3-2
3.2 DCU-4 CONTROLLING SOFTWARE 3-2
3.2.1 DCU-4 software overlays 3-2
3.2.1.1 ACOM overlay 3-3
3.2.1.2 CDEM overlay 3-3
3.2.1.3 DISK overlay 3-3
3.2.1.4 ERRECK overlay 3-4
3.2.1.5 Disk interrupt answering subroutine . 3-4
3.2.1.6 Disk driving subroutines 3-4
3.2.2 DCU-4 tables and packet structure 3-4
3.2.3 Stepflow for DCU-4 disk write request from mainframe 3-4
3.2.4 Stepflow for DCU-4 disk read request from mainframe 3-6
3.2.5 Local handling of disk queues 3-8
3.2.6 DCU-4 disk read-ahead 3-9
3.2.6.1 Disk read 3-10
3.2.6.2 Disk write 3-12
3.2.7 On-line disk diagnostic requests 3-13
3.3 DCU-4 DISK ERROR RECOVERY 3-13
3.3.1 Disk errors requiring recovery 3-14
3.3.1.1 Data error 3-15
3.3.1.2 Lost data errors 3-16
3.3.1.3 Seek errors 3-16
3.3.1.4 ID errors 3-16
3.3.1.5 Interlock status 3-16
3.3.1.6 Miscellaneous disk errors 3-17
3.3.2 I/O time-out 3-18
3.3.3 Error recovery summary 3-18
3.3.4 Error status returned to mainframe 3-20
3.3.5 DCU-4 disk error message 3-21
3.4 DCU-5 DISK CONTROLLING SOFTWARE 3-21
3.4.1 DCU-5 software components 3-22
3.4.2 DCU-5 disk driver tables and packets 3-22
3.4.2.1 Disk Request Packet (DAL) - DL@ . . . 3-22
3.4.2.2 Disk Control Block (DCB) - DK@ 3-22
3.4.2.3 Local Buffer entry - LB@ 3-22
3.4.2.4 Buffer Memory Control Block (MCB) - CB@ 3-23
3.4.2.5 Data Transfer Request (DTR) - TR@ . . 3-23
3.4.2.6 Abort Transfer Request (ATR) - AR@ . . 3-23
3.4.2.7 Device Parameter Table (DPT) - DP@ . . 3-23
3.4.2.8 MEMIO Queue Table - MEM@ 3-23
3.4.3 Resource management 3-23
3.4.3.1 Local Memory management 3-24
3.4.3.2 Buffer Memory management 3-24

3.4 DCU-5 DISK CONTROLLING SOFTWARE (continued)
3.4.4 DCU-5 disk read request stepflow 3-24
3.4.5 DCU-5 disk write request stepflow 3-25
3.4.6 DCU-5 read-ahead and write-behind 3-26
3.4.6.1 DCU-5 read-ahead 3-26
3.4.6.2 DCU-5 write-behind 3-27
3.4.7 Spiral formatting 3-28
3.4.8 On-line disk diagnostics requests 3-28
3.5 DCU-5 DISK ERROR RECOVERY 3-29
3.5.1 Recovery activity 3-30
3.5.2 Error recovery process 3-33
3.5.2.1 Unit select process 3-33
3.5.2.2 Cylinder select process 3-34
3.5.2.3 Head select-LMA select-read process . 3-35
3.5.2.4 Head select-LMA select-write process . 3-36
3.5.2.5 Unit release process 3-37
3.5.3 Operator messages 3-37
3.5.4 Error reporting 3-39
3.6 STRIPED DISK GROUPS 3-40
3.6.1 Logical to physical address mapping 3-41
3.6.2 Stepflow for a request to a striped group . . . 3-42
3.7 KERNEL INTERNAL DISK I/O 3-45

4. TAPE EXEC 4-1

4.1 ARCHITECTURE 4-1
4.1.1 Tape Exec activity 4-2
4.1.2 BYPASS activity 4-2
4.1.3 Data Stream Control Table 4-3
4.1.4 TDEM1 activity 4-4
4.1.5 Tape error recovery activities 4-4
4.2 REQUEST AND RESPONSE PACKET ROUTING 4-5
4.3 REQUEST PROCESSING 4-5
4.3.1 Configuration change request (FC$CHNGE) 4-6
4.3.2 Mount request (FC$MOUNT) 4-6
4.3.3 Read request (FC$READ) 4-9
4.3.4 Write request (FC$WRITE) 4-23
4.3.5 End read requests (FC$EOFR, FC$EORR, FC$EODR) . 4-37
4.3.6 NO-OP request (FC$NOOP) 4-40
4.3.7 Positioning requests (FC$FWFIL, FC$FWSPC, FC$BKFIL, FC$BKSPC) 4-42
4.3.8 Load display request (FC$DSP) 4-45
4.3.9 Remount request (FC$RMNT) 4-48
4.3.10 Rewind requests (FC$REWND, FC$RWND1, FC$RWND2) . 4-51
4.3.11 Unload requests (FC$UNLC, FC$UNLD1, FC$UNLD2) . 4-54
4.3.12 Free requests (FC$FREE) 4-57
4.4 ERROR RECOVERY PROCESSING 4-60
4.4.1 TAPERR routine 4-60
4.4.2 TERROR routine 4-60
4.4.3 TCART routine 4-61

4.4 ERROR RECOVERY PROCESSING (continued)
4.4.4 Recovery subroutines 4-62
4.4.4.1 Equipment check (noncartridge device only) 4-62
4.4.4.2 Bus-out check (noncartridge device only) 4-62
4.4.4.3 Intervention required (noncartridge device only) 4-62
4.4.4.4 Command reject, data converter check, and not capable 4-63
4.4.4.5 Data overrun (noncartridge device only) 4-63
4.4.4.6 Load point 4-63
4.4.4.7 Data check 4-63
4.4.4.8 Data security erase 4-63
4.4.4.9 ID burst check (noncartridge device only) 4-64
4.4.5 Error display 4-64

5. BLOCK MULTIPLEXER CHANNEL INTERFACE 5-1

5.1 IOS BLOCK MUX (BMX) SUBSYSTEM OVERVIEW 5-1
5.2 BMX CONFIGURATION 5-3
5.3 BMX TABLES 5-3
5.4 CHANNEL PROGRAM WORD (CPW) 5-9
5.4.1 Nondata transfer commands 5-9
5.4.2 Local Memory data transfer commands 5-10
5.4.3 Buffer Memory data transfer commands 5-10
5.4.4 Command chaining (CPN@CC) 5-11
5.5 DESCRIPTION OF ROUTINES 5-12
5.5.1 BMXCON 5-12
5.5.1.1 Channel configuration (CON$CHN) . . . 5-13
5.5.1.2 Control unit configuration (CON$CUT) . 5-13
5.5.1.3 Device configuration (CON$DEV) 5-14
5.5.1.4 BMXCON messages 5-14
5.5.2 BMXCPU 5-15
5.5.3 BMXSIO 5-16
5.5.3.1 Start I/O (RQ$SIO) 5-16
5.5.3.2 Wait I/O (RQ$WIO) 5-20
5.5.3.3 Return to caller 5-20
5.5.4 BMXAIO 5-20
5.5.4.1 Halt I/O (RQ$HIO) 5-21
5.5.4.2 Assign device path (RQ$APTH) 5-21
5.5.4.3 Release device path (RQ$RPTH) 5-21
5.5.4.4 Request reset (RQ$RSET) 5-21
5.5.5 BMXDEM 5-21
5.5.5.1 Start command sequence (KIC$SC) . . . 5-22
5.5.5.2 Advance command sequence (KIC$AC) . . 5-24
5.5.5.3 Advance data sequence (KIC$AD) 5-25
5.5.5.4 Request-in sequence (KIC$ER) 5-25

5.5 DESCRIPTION OF ROUTINES (continued)
5.5.6 BMX interrupt handler (IBMX) 5-26
5.5.6.1 Immediate return (KIC$IR) 5-27
5.5.6.2 Advance data (KIC$AD) 5-27
5.5.6.3 Start request-in (KIC$SR) 5-27
5.5.6.4 Continue request-in (KIC$CR) 5-28
5.5.6.5 End request-in (KIC$ER) 5-28
5.5.7 BMXOPE 5-28
5.5.7.1 Open (FC$MOUNT/FC$REMOUNT) 5-28
5.5.7.2 Close (FC$FREE) 5-28
5.5.8 BMXTPO 5-29

6. I/O SUBSYSTEM STATION 6-1

6.1 STATION TASKS 6-1
6.2 STATION STORAGE 6-2
6.3 TASK FLOW AND INTERACTION 6-5
6.3.1 Station initialization 6-5
6.3.2 KEYBD task 6-6
6.3.3 DISPLAY task 6-8
6.3.4 CLI task 6-11
6.3.5 PROTOCOL task 6-16
6.3.6 STAGEIN task 6-25
6.3.7 STAGEOUT task 6-27
6.3.8 STIO overlay 6-29
6.3.9 POST overlay 6-32
6.4 GLOBAL SYMBOLS 6-33
6.5 CONSOLE OUTPUT 6-34
6.6 SCREEN IMAGE 6-34

7. FRONT-END CONCENTRATOR 7-1

7.1 CONC OVERLAY DESCRIPTION (CONCENTRATOR INITIALIZATION) . 7-2
7.2 CONCIO ACTIVITY DESCRIPTION 7-2
7.3 CONCID OVERLAY DESCRIPTION 7-7
7.4 CONCERR OVERLAY DESCRIPTION 7-7
7.5 ENDCONC OVERLAY DESCRIPTION 7-8

8. INTERACTIVE STATION 8-1

8.1 INTERACTIVE CONCENTRATOR OVERLAYS 8-1
8.1.1 IAIOP overlay 8-1
8.1.2 IAIOP1 overlay 8-4
8.1.3 IAFUNC overlay 8-4
8.1.4 IAMSG overlay 8-5

8. INTERACTIVE STATION (continued)

8.2 INTERACTIVE CONSOLE OVERLAYS 8-6
8.2.1 IACON overlay 8-6
8.2.2 IACON1 overlay 8-7
8.2.3 IACMD overlay 8-7
8.2.4 IAOUT overlay 8-8

9. USER-CHANNEL I/O 9-1

9.1 USER CHANNEL REQUESTS 9-1
9.1.1 Open request (CR$OPN) 9-1
9.1.2 Read request (CR$RD) 9-2
9.1.3 Read-hold request (CR$RDH) 9-2
9.1.4 Read-read request (CR$RD2) 9-2
9.1.5 Write request (CR$WRT) 9-2
9.1.6 Write-hold request (CR$WRTH) 9-3
9.1.7 Write-write request (CR$WRT2) 9-3
9.1.8 Driver request (CR$DRV) 9-3
9.1.9 Close request (CR$CLS) 9-3
9.2 SHELL ARCHITECTURE 9-4
9.2.1 User Channel Table 9-5
9.2.2 User channel message handler 9-5
9.2.3 User channel shell (UCSHL) 9-5
9.2.3.1 UCSHL open subroutine (UCOPN) 9-5
9.2.3.2 UCSHL close subroutine (UCCLS) 9-6
9.2.3.3 UCSHL read subroutine (UCRD) 9-6
9.2.3.4 UCSHL write subroutine (UCWRT) 9-6
9.2.3.5 UCSHL driver subroutine (UCDRV) . . . 9-7
9.2.4 User channel shell data handler (UCXFR) 9-7
9.3 SHELL AND DRIVER INTERFACE 9-7
9.3.1 SIGNAL and WATCH macros 9-7
9.3.2 Shell requests 9-8
9.3.3 Driver responses 9-9
9.3.4 Buffering 9-9
9.3.5 Interrupt processing 9-10
9.3.6 User channel configuration 9-10
9.3.7 Driver installation 9-10

10. NSC HYPERCHANNEL 10-1

10.1 NSC ACTIVITY INITIALIZATION 10-1
10.2 NSCIO ACTIVITY 10-2
10.2.1 NSCIO idle loop 10-2
10.2.2 Write sequence for the protocol-independent interface 10-3
10.2.3 Read sequence for the protocol-independent interface 10-3
10.2.4 SCP interface logon sequence 10-4

10. NSC HYPERCHANNEL (continued)

10.3 NSC ACTIVITY TERMINATION 10-6
10.4 OVERLAYS 10-6
10.4.1 ADEM overlay 10-6
10.4.2 FNSC overlay 10-6
10.4.3 NIDEND overlay 10-9
10.4.4 NSC overlay 10-9
10.4.5 NSCEND overlay 10-9
10.4.6 NSCID overlay 10-9
10.4.7 NSCIO overlay 10-9
10.4.8 NSCMSG overlay 10-9
10.4.9 NSCRW overlay 10-10
10.4.10 SCPIO overlay 10-10
10.4.11 TERMNSC overlay 10-10
10.5 ERROR RECOVERY 10-10
10.5.1 Error recovery for SCP protocol 10-11
10.5.1.1 Driver input/read operations 10-11
10.5.1.2 Driver output/write operations 10-11
10.5.2 Error recovery for the protocol-independent interface 10-12
10.5.2.1 Driver input/read operations 10-12
10.5.2.2 Driver output/write operations 10-12
10.6 CHANNEL/ID ORDINAL DESCRIPTION 10-12

11. FRONT-END INTERFACE LOGICAL PATH ACTIVITY 11-1

11.1 FEI LOGICAL PATH ACTIVITY INITIALIZATION 11-1
11.2 FEI LOGICAL PATH ACTIVITY TERMINATION 11-1
11.3 OVERLAYS 11-2
11.3.1 ADEM overlay 11-2
11.3.2 FNSC overlay 11-3
11.3.3 FEIR overlay 11-3
11.3.4 FEIW overlay 11-3
11.3.5 FEIMSG overlay 11-3

12. HSX CHANNEL INTERFACE 12-1

12.1 HSX CHANNEL REQUESTS 12-1
12.1.1 OPEN request (HSF$OPEN) 12-2
12.1.2 READ request (HSF$READ) 12-2
12.1.3 WRITE request (HSF$WRIT) 12-2
12.1.4 CONTROL request (HSF$CNTL) 12-2
12.1.4.1 Set parameters (HSS$SET) 12-2
12.1.4.2 Send interrupt (HSS$SNDI) 12-3
12.1.4.3 Receive interrupt (HSS$RECI) 12-3
12.1.5 CLOSE request (HSF$CLOS) 12-3

12. HSX CHANNEL INTERFACE (continued)

12.2 HSX DRIVER ARCHITECTURE 12-3
12.2.1 HSX DEMON overlay (HCOM) 12-3
12.2.2 HSX input interrupt handler (HSXI) 12-4
12.2.3 HSX output interrupt handler (HSXO) 12-4
12.2.4 Buffering 12-4
12.3 DEBUG MODE 12-5
12.4 OVERLAY LISTING 12-5
12.5 ERROR PROCEDURES 12-5
12.5.1 Input errors 12-6
12.5.1.1 Clear pulse received (HST$CLR) 12-6
12.5.1.2 Multiple bit error (HST$DATA) 12-6
12.5.1.3 Data overrun error (HST$OVER) 12-6
12.5.1.4 Long block error (HST$LONG) 12-7
12.5.1.5 Software time-out (HST$TMO) 12-7
12.5.1.6 Device not present (HST$NDEV) 12-7
12.5.1.7 Short block error (HST$SHRT) 12-7
12.5.2 Output errors 12-7
12.5.2.1 Exception pulse received during transfer (HST$XDT) 12-8
12.5.2.2 Exception pulse received while channel idle (HST$XFT) 12-8
12.5.2.3 Receiving device aborted (HST$ABRT) . 12-8
12.5.2.4 Software time-out (HST$TMO) 12-8
12.5.2.5 Device not present (HST$NDEV) 12-8
12.6 SPECIAL SEQUENCES 12-9
12.6.1 Input sequences 12-9
12.6.1.1 Send exception pulse (HSS$SNDI) . . . 12-9
12.6.1.2 Wait for clear pulse (HSS$RECI) . . . 12-9
12.6.2 Output sequences 12-9
12.6.2.1 Send clear pulse (HSS$SNDI) 12-10
12.6.2.2 Wait for exception pulse (HSS$RECI) . 12-10

13. VMEBUS (FEI) DRIVER 13-1

13.1 N-PACKET INTERFACE 13-1
13.2 DRIVER OVERLAYS 13-2
13.2.1 ADEM overlay 13-2
13.2.2 FNSC overlay 13-2
13.2.3 NSCRW overlay 13-4
13.2.4 VME overlay 13-4
13.2.5 VMEND overlay 13-4
13.2.6 FEIMSG overlay 13-4
13.2.7 VMERD overlay 13-4
13.2.8 VMEWT overlay 13-5
13.2.9 TERMVME overlay 13-5
13.2.10 TERMNSC overlay 13-5
13.2.11 NSCID overlay 13-5
13.2.12 SCPIO overlay 13-5

13. VMEBUS (FEI) DRIVER (continued)

13.3 READ AND WRITE REQUESTS FLOW DESCRIPTIONS 13-5
13.3.1 Read request sequence 13-6
13.3.2 Write request sequence 13-7
13.4 FLOW DESCRIPTION FOR SCP PROTOCOL 13-7
13.5 INTERRUPT HANDLING 13-8

14. PROGRAM LIBRARY AND MACROS 14-1

14.1 PL STRUCTURE 14-1
14.1.1 Common deck structure 14-2
14.1.2 Adding an overlay 14-2
14.2 MACROS 14-4
14.2.1 Exit stack macros 14-8
14.2.1.1 EGET macro 14-9
14.2.1.2 EPUT macro 14-9
14.2.1.3 EINCR macro 14-9
14.2.1.4 EDECR macro 14-10
14.2.1.5 EXSGET macro 14-10
14.2.1.6 EXSPUT macro 14-10
14.2.2 Execution control macros 14-11
14.2.2.1 $IF macro 14-11
14.2.2.2 $UNTIL macro 14-13
14.2.2.3 $GOTO macro 14-14
14.2.2.4 $PUNTIF macro 14-15
14.2.3 Data definition macros 14-16
14.2.3.1 FIELD macro 14-16
14.2.3.2 ISFIELD macro 14-17
14.2.3.3 TABLE macro 14-18
14.2.4 Data access macros 14-19
14.2.4.1 ADDRESS macro 14-19
14.2.4.2 GET macro 14-20
14.2.4.3 LOAD macro 14-20
14.2.4.4 PUT macro 14-22
14.2.4.5 STORE macro 14-22
14.2.4.6 RGET macro 14-22
14.2.4.7 RPUT macro 14-23
14.2.4.8 RSTORE macro 14-23
14.2.4.9 FLDADD macro 14-25
14.2.4.10 FLDSUB macro 14-25
14.2.5 Overlay and register definition macros 14-26
14.2.5.1 OVERLAY macro 14-26
14.2.5.2 REGDEFS macro 14-27
14.2.5.3 REGISTER macro 14-28
14.2.5.4 RETREG macro 14-29
14.2.6 Memory macros 14-31
14.2.6.1 CLEAR macro 14-31
14.2.6.2 COPY macro 14-31

15. DEBUGGING TOOLS 15-1

15.1 SUMMARY UTILITY 15-1
15.1.1 Overlays 15-3
15.1.2 Interrupts 15-3
15.1.3 Kernel calls 15-4
15.2 HISTORY TRACE 15-4
15.2.1 Examining trace buffers on-line 15-4
15.2.2 Examining trace buffers off-line 15-7
15.2.3 Trace event codes, subcodes, and parameters . . 15-8
15.3 DEBUGGER 15-22
15.3.1 Display accumulator command 15-24
15.3.2 Display B register command 15-24
15.3.3 Display carry register command 15-25
15.3.4 Display channel status command 15-25
15.3.5 Issue a function on a channel command 15-25
15.3.6 Display exit stack command 15-26
15.3.7 Display operand register command 15-26
15.3.8 Toggle display mode command 15-27
15.3.9 Display Local Memory command 15-27
15.3.10 Display P register command 15-28
15.3.11 Set single breakpoint command 15-28
15.3.12 Set double breakpoint command 15-28
15.3.13 Display breakpoints command 15-29
15.3.14 Delete breakpoints command 15-29
15.3.15 Set count register and proceed from breakpoint command 15-29
15.3.16 Display Buffer Memory command 15-30
15.3.17 Display high-speed channel command 15-31
15.3.18 Processing of channels used by the debugger command 15-31
15.4 PATCH OVERLAY 15-32
15.5 LISTP OVERLAY 15-33
15.6 LISTO OVERLAY 15-34
15.7 DKDMP OVERLAY 15-36

APPENDIX SECTION

A. DUMP ANALYSIS A-1

B. IOS CONFIDENCE UTILITIES B-1

B.1 CHNTEST command B-2
B.2 CPTEST command B-3
B.3 ECHOCP command B-4
B.4 HSPTEST command B-5
B.5 MOSTEST command B-7
B.6 SSDTEST command B-9

B. IOS CONFIDENCE UTILITIES (continued)

B.7 STOP command B-11
B.8 XDK command B-11
B.9 XMT command B-12
B.10 XPR command B-13

C. SYSDUMP C-1

C.1 OPERATIONAL DESCRIPTION C-1
C.2 DUMP FORMAT C-8

D. ISP CHANNEL DRIVER D-1

D.1 MAIN LOOP D-1
D.2 OPEN PROCESSING D-1
D.3 CLOSE PROCESSING D-2
D.4 I/O REQUEST PROCESSING D-2
D.5 STARTIO SUBROUTINE D-3
D.6 WAITIO SUBROUTINE D-3
D.7 GETL SUBROUTINE D-4

FIGURES

1-1 A Cray Computer System with Four I/O Processors 1-4
2-1 Local Memory Structure 2-2
2-2 Buffer Memory Organization 2-4
2-3 ALERT Stepflow 2-17
2-4 AWAKE Stepflow 2-20
3-1 Striped Group (Six Physical Units Constituting One Logical Unit) 3-41
3-2 Target Memory Mapping for a Single Device 3-44
3-3 Target Memory Mapping for a Two-unit Group 3-44
4-1 Processing of Configuration Change Requests 4-7
4-2 Processing of Mount Requests 4-8
4-3 Processing of Read Requests 4-9
4-4 Processing of Write Requests 4-24
4-5 Processing of End Read Requests 4-38
4-6 Processing of No-op Requests 4-40
4-7 Processing of Positioning Requests 4-42
4-8 Processing of Display Requests 4-46
4-9 Processing of a Remount Request to the Same Device 4-49
4-10 Processing of a Remount Request to a New Device 4-50
4-11 Processing of Rewind Requests 4-51
4-12 Processing of Unload Requests 4-55
4-13 Processing of Free Requests 4-58
5-1 BMX Overview 5-2
5-2 A 2-by-2 Configuration (Multiple Path, Single Bank) 5-4

FIGURES (continued)

5-3 Two 1-by-1 Configurations (Single Path, Multiple Bank) 5-5
5-4 A 2-by-1 Configuration (Multiple Path, Multiple Bank) 5-6
5-5 Pointer to Channel Tables for Each Configured Channel 5-7
5-6 Pointer to Device Table for Each Configured Device 5-8
5-7 BMXDEM's Usage of the Channel Table 5-22
5-8 Location of BDV@UN 5-26
6-1 Local Memory Stack Area 6-4
6-2 Station Initialization Flow 6-6
6-3 KEYBD Task Flow and Interaction 6-7
6-4 DISPLAY Task Flow and Interaction Operator Displays 6-11
6-5 CLI Task Flow and Interaction 6-13
6-6 STATUS Command Flow and Interaction 6-14
6-7 DROP Command Flow 6-15
6-8 PROTOCOL Task Flow (Initialization) 6-19
6-9 PROTOCOL Task Flow and Interaction (Main Body) 6-20
6-10 PROTOCOL Task Flow (Termination) 6-24
6-11 STAGEIN Task Flow and Interaction 6-26
6-12 STAGEOUT Task Flow and Interaction 6-28
7-1 Tree Structure of Concentrator Software 7-1
8-1 Structure of Interactive Concentrator Software 8-2
8-2 Structure of Interactive Console Software 8-2
9-1 Shell Architecture 9-4
10-1 NSC HYPERchannel Driver Overlay Connection 10-7
11-1 FEI Logical Path Overlay Connections 11-2
13-1 VMEbus Driver Overlay Connections 13-3
15-1 The SUMMARY Display 15-2
15-2 History Trace Sample Output 15-6
15-3 LISTO Sample Output 15-32
C-1 SYSDUMP Memory Map of IOP with Master Disk Attached C-2
C-2 AI List C-3
C-3 Head Format for OS = UNICOS C-4

TABLES

2-1 System Directory Contents 2-5
2-2 Overlay Format 2-11
2-3 Summary of Service Functions 2-14
2-4 I/O Processor Intercommunication Function Codes 2-53
3-1 Error Conditions 3-14
3-2 Interlock Error Conditions 3-17
3-3 Miscellaneous Error Conditions 3-18
3-4 Disk Error Recovery Summary 3-19
3-5 Disk Error Information in DAL 3-20
3-6 DD-49 Error Retry Limits 3-30
3-7 RD-10, DD-39, and DD-40 Error Retry Limits 3-31
6-1 Station Tasks 6-2
6-2 Shared Memory Access 6-3
6-3 KEYBD Task Interaction Areas 6-7

TABLES (continued)

6-4 DISPLAY Task Interaction Areas 6-8
6-5 PROTOCOL Task Interaction Areas 6-16
6-6 STAGEIN Task Interaction Areas 6-25
6-7 STAGEOUT Task Interaction Areas 6-27
14-1 Summary of Macros 14-5
15-1 Trace Event Codes 15-7
15-2 Trace Event Parameters 15-9
A-1 Kernel Registers A-3
D-1 Stepflow for DOIO Buffered Loops D-2

INDEX

1. INTRODUCTION

This manual describes the internal design of the software running in the I/O Subsystem (IOS) of the CRAY Y-MP, CRAY X-MP EA, CRAY X-MP, and CRAY-1 computer systems. IOS software supports either the COS or UNICOS operating systems.

The parts of IOS software are as follows (with references to the section that describes them):

- The Kernel (section 2)
- Disk I/O (section 3)
- Tape Exec (section 4)
- Block multiplexer channel interface (section 5)
- IOS station (section 6)
- Front-end concentrator (section 7)
- Interactive station (section 8)
- User channel I/O (section 9)
- NSC HYPERchannel (section 10)
- Front-end interface logical path activity (section 11)
- HSX channel interface (section 12)
- VMEbus (FEI-3) driver (section 13)
- Program library and macros (section 14)
- Debugging tools (section 15)

Commands for the IOS are described in the I/O Subsystem (IOS) operator's guides.

The Kernel serves as the operating system. A copy of the Kernel runs in each I/O Processor (IOP) in the subsystem, adapting itself to the special functions of each processor. In addition to the operating system section, Kernel software includes the following:

- A deadstart package
- An interactive debugger
- A buffer from which overlay areas are allocated
- A section of free memory
- An I/O buffer area
- A trace buffer area

The disk I/O software moves data in streams between Central Memory in the mainframe or SSD Memory in the optional SSD solid-state storage device, and disks attached to the IOS.

Tape Exec (TEX) software processes requests from the mainframe. TEX performs functions related to tape I/O such as message routing, data formatting, data movement, and error recovery.

Block multiplexer channel interface software drives the block multiplexer channel hardware. It contains device-independent command and interrupt code that executes requests from the Tape Exec.

Station software runs in the Master I/O Processor (MIOP) and supports operator commands, station displays, and dataset staging. All dataset staging is performed without queuing; datasets are transferred directly to the mainframe.

Concentrator software accepts data from front-end computers into IOS Local Memory, builds the data into a message, and sends it to the mainframe. The concentrator relieves the mainframe of the burden of handling an interrupt for each subsegment of messages transferred.

The interactive station permits interaction with a job running in the mainframe. Interactive commands are entered at a console connected directly to the IOS.

User Channel I/O software runs in the MIOP and supports access to IOS channels by COS tasks. User Channels may be used for connecting new devices or mainframes to the IOS.

The NSC HYPERchannel driver links a Cray mainframe and a front-end through the NSC HYPERchannel. The driver allows multiple front-end computers to be connected to one physical MIOP channel pair.

The Front-end Interface (FEI) logical path driver provides an FEI connection for UNICOS. This connection parallels the NSC logical path connection. The driver allows front-end stations to communicate with the UNICOS Station Call Processor (USCP) under UNICOS by using the SCP protocol.

The HSX High-speed External Communications channel driver supports the CRI HSX channel.

The VMEbus driver allows a VMEbus-based front-end processor connected to a CRI VMEbus interface to communicate with a Cray computer system. The driver allows simultaneous use of multiple application protocols.

The IOS software program library (IOPPL) contains the following: the system text ($APTEXT), the Kernel, the configuration overlay AMAP, overlay decks, TAPELOAD, DISKLOAD, DUMP, and CAL overlays used for deadstarting and dead dumping the mainframe. Macro instructions used by the IOS are defined in $APTEXT and perform exit stack access, execution control, table access, and overlay and register definition.

The debugging tools provide a means to analyze and maintain IOS software.

1.1 HARDWARE SPECIFICATIONS

The IOS consists of channel interfaces, at least one-half million words of Buffer Memory, and two, three, or four IOPs.

Each IOP contains a Local Memory section, a computation section, a control section, and an I/O section.

The computation section includes functional units for integer arithmetic (addition and twos complement subtraction) and shifting. The computation section does not perform multiplication, division, or floating-point arithmetic. The accumulator (A register) is a 16-bit register used in single-address operations. Each IOP has 512 16-bit operand registers.

The control section consists of an instruction stack, a program exit stack, and control logic. The instruction stack is a 32-parcel circular buffer. New instructions brought in from memory replace the parcels that have resided the longest in the instruction stack. The program exit stack is a set of 16 registers that stores return addresses during the execution of subroutines and the Kernel interrupt processing routine. Local Memory contains 65,536 parcels of 16 bits, plus 2 parity bits, each. Local Memory is located in four sections, each consisting of four banks.

The IOS model B is linked to the mainframe by two types of channels: the 100-Mbyte channel used for data streaming and the 6-Mbyte channel used to pass control information. The standard configuration for the IOS model B provides one channel of each type (the 6-Mbyte channel connected to the MIOP, the 100-Mbyte channel connected to the Buffer I/O Processor (BIOP)); a second 100-Mbyte channel linking the Disk I/O Processor (DIOP) and the mainframe is optional. A detailed description of the IOS model B is contained in the I/O Subsystem Model B Hardware Reference Manual, publication HR-0030.

The IOS model C is linked to the mainframe with the same type of channels, but its standard configuration contains one channel of each type for each IOP. In addition, the 100-Mbyte channels on the model C supports a Bypass mode of operation that enables data transfer directly between Buffer Memory and Central Memory bypassing the IOP's Local Memory. A detailed description of the IOS model C is contained in the I/O Subsystem Model C Hardware Reference Manual, publication HR-0081.

The IOS may optionally be linked to an SSD Memory by a 100-Mbyte channel. Software supports the movement of disk data over such a channel if it is attached to the BIOP, an optional DIOP, or an optional Auxiliary I/O Processor (XIOP). This channel can support the Bypass mode of operation on a model C IOS if attached to channels 14_8 and 15_8 on an IOP.

1.2 SYSTEM CONFIGURATION

The IOS has a minimum of two IOPs. The MIOP and BIOP are mandatory. In addition, the IOS can have either one or two DIOPs, permitting a maximum of 48 disk units to be connected to the system. One XIOP, which controls block multiplexer channels, can be selected as the third or fourth processor. Figure 1-1 shows an example of a Cray computer system with each of the four types of IOPs.

Each processor in the IOS is responsible for a unique set of functions. A processor's functions are defined by the peripheral equipment attached to it. The software in each processor knows its functions and is structured to perform these functions as efficiently as possible. The processors can communicate with each other through Buffer Memory. Thus, a processor can request that another processor perform a function for it.

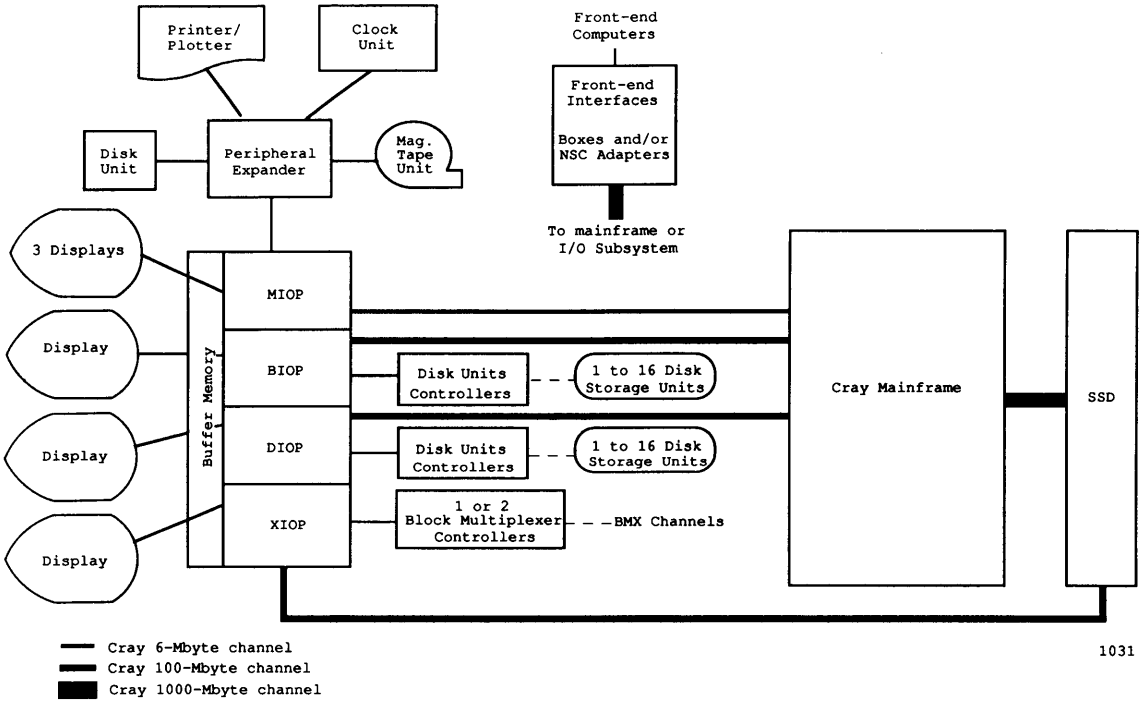


Figure 1-1. A Cray Computer System with Four I/O Processors

MIOP responsibilities are as follows:

- The MIOP is the first IOP in the IOS to be deadstarted. The MIOP initializes the contents of Buffer Memory and deadstarts the other processors in the configuration using Buffer Memory and accumulator channels to the other processors.
- The MIOP and the BIOP are used to deadstart the mainframe.
- The MIOP handles all communication with the mainframe over a 6-Mbyte channel. This traffic includes disk and tape requests and station communications.
- The MIOP performs front-end and station software support.
- The MIOP handles input and output operations on the expander channel.
- The MIOP accepts information from the error channel and transmits it to the mainframe for inclusion in the system error log.
- The MIOP is the operator interface to the IOS editor, which maintains deadstart and restart parameter files.
- The MIOP handles input and output operations on user channels for COS tasks.

BIOP responsibilities are as follows:

- The BIOP uses a 100-Mbyte channel to transfer data between Central Memory and Buffer Memory for all IOPs.
- The BIOP transfers tape data between Central Memory and Buffer Memory under direction of the XIOP. It also blocks and deblocks tape data as it is moved between Central Memory and Buffer Memory.
- The BIOP performs disk I/O to and from disk units attached to its channels. (IOS software supports the DD-19, DD-29, DD-39, DD-40, RD-10, and DD-49 Disk Storage Units.) It performs error recovery when errors are detected on data transfers.
- If a 100-Mbyte channel is connected to SSD Memory, the BIOP transfers disk data between SSD Memory and Local Memory.

DIOP responsibilities are as follows:

- The DIOP moves data from Buffer Memory to disk and vice versa at the request of packets from the mainframe through the MIOP. These packets also return status to the requester.

- When errors are detected in data transfers to or from disk, DIOP software attempts error recovery.

- If a 100-Mbyte channel is connected to Central Memory, the DIOP transfers data between Central Memory and Local Memory.

- If a 100-Mbyte channel is connected to SSD Memory, the DIOP transfers disk data between SSD Memory and Local Memory.

XIOP responsibilities are as follows:

- The XIOP handles data to and from IBM-compatible tape drives and buffers the data to and from Buffer Memory at the request of packets from the mainframe.

- When errors are detected while transferring data to or from tape, the XIOP performs error recovery procedures.

- If a 100-Mbyte channel is connected to SSD Memory, the XIOP transfers disk data between SSD Memory and Local Memory.

Each processor logs information and keeps statistics about channel use, error detection, and error recovery.

1.3 REGISTER ASSIGNMENTS

The 512 operand registers are conventionally assigned to IOS software entities as follows (register numbers are in octal).

Registers	Software Entity
0-377	Kernel
400-577	Overlays
600-677	Interrupt handling overlays
700-777	Debug packages (the debugger, trace, and DUMP)

The % symbol usually designates global Kernel registers and R! usually designates APML assembly operand registers.

1.4 TERMINOLOGY

Although the IOPs are usually referred to by their acronyms (MIOP, BIOP, DIOP, and XIOP), they can also be referred to as IOPs 0 through 3. In this manual, the third and fourth IOPs are often referred to as IOP-2 and IOP-3.

This terminology is necessary because the identities of those two processors can vary. If the IOS has three IOPs, either IOP-2 or IOP-3 can be present. If the IOS has four processors, both IOP-2 and IOP-3 are present. IOP-2, when present, must be a DIOP due to hardware limitations. IOP-3, when present, can be a DIOP or an XIOP. When IOP-3 is specified, the documentation applies to either a DIOP or an XIOP.

The words *task* and *activity* are used interchangeably in this manual. Activities (or tasks) are routines that do specific jobs within the subsystem.

An *activity* is initiated either by a command keyed in at a Kernel console or by a Kernel service request function. Within the domain of an activity there is a set of routines that reside as overlays in Buffer Memory. These routines operate under a stack-like structure. When an activity begins, an initial routine gains control. The routine may give control to another routine, and so on. As a routine completes, control may pass back to the previous routine. An activity is terminated when the initial routine relinquishes control.

Parcel is used in this manual when referring to a storage unit of 16 bits. *Word* refers to a 64-bit storage unit.

Central Memory consists of 64-bit words and is located in the mainframe. *SSD Memory* consists of 64-bit words and is located in an optional SSD solid state storage device. Data transfer to or from SSD Memory must be a multiple of 64 words, and to an address that is a multiple of 64 words. *Buffer Memory* consists of 64-bit words and is located in the IOS chassis. *Local Memory* stores information in 16-bit units (parcels) and is located within each IOP.

Target memory refers to the ability of the mainframe operating system to specify the source or destination of disk data as being Central Memory, SSD Memory, or the portion of Buffer Memory reserved for dataset storage. An IOP with a 100-Mbyte channel connected to a particular target memory is referred to as the *Target Memory Processor* for that memory type.

Although the IOS includes 100-Mbyte channels between each of the IOPs and Buffer Memory, the term *100-Mbyte channel* as used in this manual refers only to the channel linking an IOP to Central Memory or SSD Memory. Similarly, *6-Mbyte channel* refers to the command channel linking an IOP to the mainframe, unless otherwise stated.

1.5 FORMAL SYNTAX CONVENTIONS

This manual uses the following conventions to describe macro calls, console commands, and other formal representations:

- An uppercase word, such as MSG, is a predefined system keyword.
- A lowercase word in italics, such as *msg*, represents variable data. Italics also highlights terms being defined.
- Information delimited by square brackets, [], is optional.
- A vertical bar in a command format (A|B) separates two or more literal parameters when only one choice can be used.

Any command entered at a console must be followed by a carriage return, unless otherwise specified.

2. THE KERNEL

The Kernel is the software package that controls activities running in each I/O Processor (IOP). Each IOP has its own copy of the Kernel, and all copies are basically the same. At deadstart, each copy of the Kernel dynamically adjusts to configurations and functions assigned to its IOP.

Kernel functions consist of answering interrupts, managing overlay areas, and handling independent activities running in the I/O Subsystem (IOS). The functions of the independent activities include handling other activities, allocating memory, and partial handling of peripherals.

2.1 LOCAL MEMORY USAGE

Kernel code, as figure 2-1 shows, is stored in a separate area of Local Memory away from the constants and tables it references. The table area contains configuration maps, memory allocation tables, activity dispatching parameters, and information about overlays in Buffer Memory and in Local Memory.

Space for overlays follows the Kernel code area in Local Memory. A specified amount of space is available to each of the IOPs and is allocated dynamically as new overlays are loaded.

Three memory areas managed by the Kernel come next in Local Memory. One area provides communication packets (DALs), one contains free memory, and one provides I/O buffers. The DAL area contains a linked list of 32-parcel packets. The free memory area is used for Kernel tables and small buffers and is organized as a chain structure. Free memory is allocated in multiples of 4 parcels, with the first address always falling on a 4-parcel boundary.

The I/O buffer area is allocated in pieces of 2048 parcels.

The relative size of each of these types of areas in Local Memory is determined by parameters in the overlay AMAP during deadstart. (See the COS Operational Procedures Reference Manual, publication SM-0043, or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018, for information on IOP configurations set in AMAP.) The size of the areas depends on the functions that each IOP performs. For example, an IOP that is used exclusively for disk I/O has more Local Memory assigned to I/O buffers than an IOP that performs functions in addition to disk I/O.

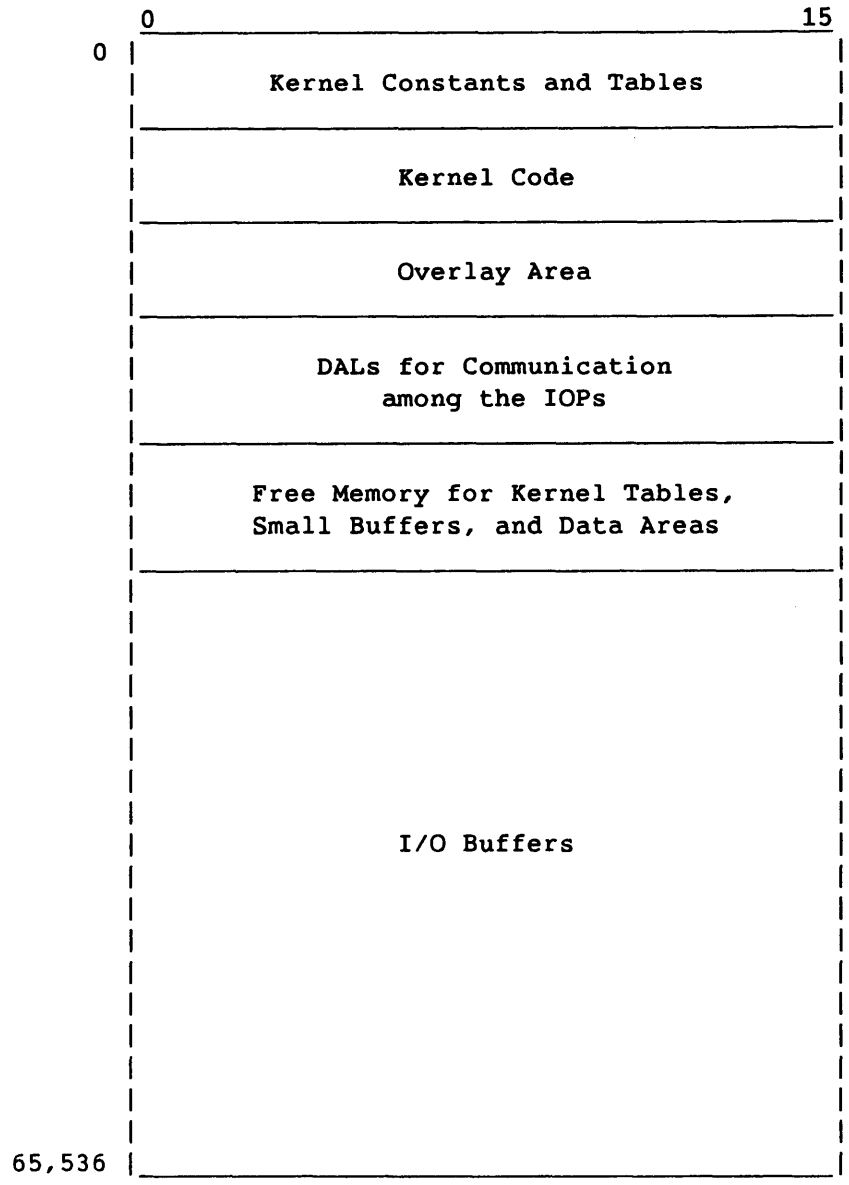


Figure 2-1. Local Memory Structure

2.1.1 LOCAL MEMORY SCRUBBING

In the IOS Model B, the ECL technology used for IOP Local Memory is susceptible to *soft* memory errors caused by alpha particle migration. These *soft* errors are seen as bits changing state in areas of memory that are infrequently or never written. Because the IOS Model B has no Local Memory correction and the IOS Model C has SECDED Local Memory error correction only during transfer of data from Local Memory to issue, a method is needed to correct Local Memory errors.

The SCRUB routine is used to correct Local Memory errors. SCRUB is created in each IOP at deadstart time. It uses the TPUSH Kernel call to activate itself once every 15 minutes.

SCRUB allocates a Buffer Memory data buffer and uses it to write out, then read back in, the Kernel-resident area of Local Memory. SCRUB then issues a FLUSH Kernel call, which causes all overlays to be reloaded from Buffer Memory before being activated. The time spent in SCRUB is approximately 0.5 ms.

The remainder of Local Memory consists of dynamically allocated parcels that are written before they are read.

2.2 BUFFER MEMORY USAGE

The IOPs share Buffer Memory, which is organized according to the plan defined in figure 2-2. The first locations are reserved for a deadstart package. During deadstart, the Master I/O Processor (MIOP) initializes common tables and the System Directory so that all the control information is ready to begin execution when the other IOPs are deadstarted.

The next area in Buffer Memory is reserved for the System Directory, followed by the message area, Kernel area (includes AMAP, the overlays, and IOPs Kernel storage), and lastly, Buffer Memory resident datasets.

Buffer Memory addresses require 32-bits (2 parcels). The high-order bits of the address are in the first parcel and the low-order bits are in the second.

2.2.1 SYSTEM DIRECTORY

The System Directory contains pointers to other information saved in Buffer Memory, including message area locations for each processor, and pointers to Kernel storage reserved for each processor. The System Directory begins at the first address after the deadstart package. Table 2-1 lists the directory structure.

All of the IOPs can access the System Directory, but information in the directory can be changed only by the MIOP during Deadstart.

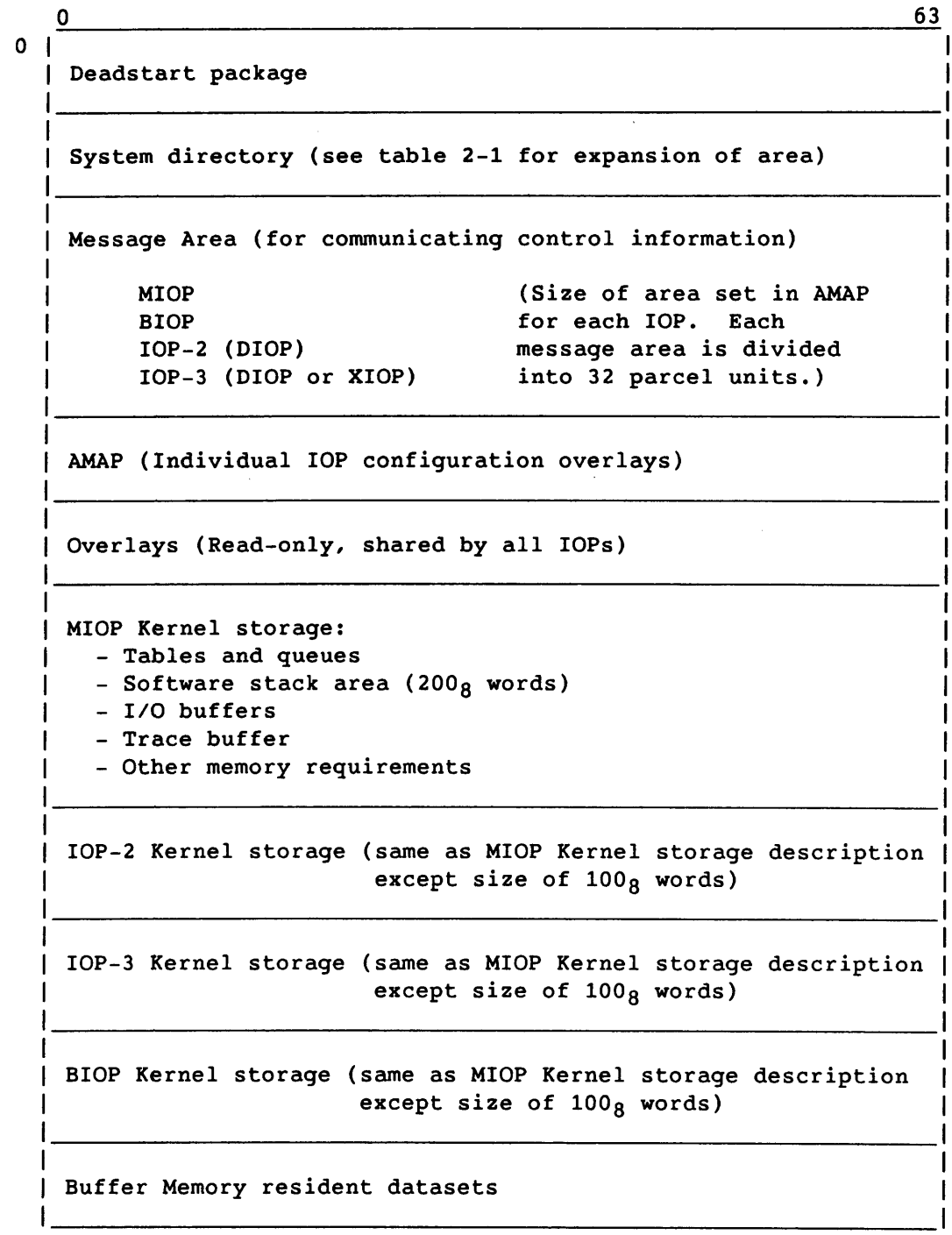


Figure 2-2. Buffer Memory Organization

Table 2-1. System Directory Contents

Parcel	Bits	Description
0-1	0-15	MIOP message area
2-3	0-15	BIOP message area
4-5	0-15	IOP-2 message area
6-7	0-15	IOP-3 message area
8-11	0-15	Reserved
12-13	0-15	First overlay (AMAP) address
14-15	0-15	Unused
16-17	0-15	MIOP Kernel storage area
18-19	0-15	Size of MIOP storage area in 512-word blocks
20-21	0-15	BIOP Kernel storage area
22-23	0-15	Size of BIOP storage area in 512-word blocks
24-25	0-15	IOP-2 Kernel storage area
26-27	0-15	Size of IOP-2 storage area in 512-word blocks
28-29	0-15	IOP-3 Kernel storage area
30-31	0-15	Size of IOP-3 storage area in 512-word blocks

2.2.2 MESSAGE AREAS

Message areas accessed by senders and receivers of messages follow the System Directory. The sending IOP maintains control of the area and allocates or deallocates memory within it. The receiving processor signals when the message has been received and processed; the memory is then released to the pool of message areas belonging to the sender. This process is described in detail in subsection 2.13, Communications Among IOPs.

2.2.3 KERNEL AREA

Each IOP has access to its own reserved Kernel storage area, which holds temporary information about activities and swapped activity areas. Reserved areas also provide data buffer storage for disks and other peripherals. A buffer is also reserved for history trace information. (TRACE is a debugging tool described in subsection 15.2, History Trace.) Each area is solely under the control of its respective IOP.

2.2.4 BUFFER MEMORY RESIDENT DATASETS

Part of Buffer Memory can be allocated for dataset storage. See the COS Operational Procedures Reference Manual, publication SM-0043, or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018, for more information about configuring storage for Buffer Memory resident datasets.

2.3 TARGET MEMORY

The IOS may be configured so that it can access various types of memory through 100-Mbyte channels. The concept of a target memory allows IOS activities to specify which of these memories to access in Kernel Service Request I/O functions. It also allows the mainframe to specify which memory type to use in disk requests. An IOP with a 100-Mbyte channel connected to a particular memory is referred to as the Target Memory Processor for that memory.

Memory types are defined in $APTEXT by equates as follows:

Memory Type	Description
TM$CMEM	Central Memory
TM$SSD	SSD Memory
TM$BMR	BMR Memory

The Target Memory Control Block (TMCB) residing in the Kernel table area contains a pointer to the Target Memory Control Table (TM@) for each type of target memory. The IOS Kernel Service Request routines and the disk software use these control tables to determine memory base and limit addresses, and to determine the Target Memory Processor number for each memory type.

2.4 ACTIVITY-SOFTWARE STACKING

Each independent activity is controlled by the Kernel using the parameters and data stored within the Activity Descriptor (AD) for that activity. The Kernel establishes an AD in the free memory section of Local Memory when a new activity is created. The AD for each activity currently in use remains in Local Memory until the activity is terminated. The format of the Activity Descriptor is defined in the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

An activity may be composed of more than one overlay. When control passes to a new overlay that is not already resident in Local Memory, the new overlay is copied from Buffer Memory to the overlay area in Local Memory, where it executes.

Each overlay to execute is assigned a storage module (SMOD), which saves information relating to that overlay's executing environment. An SMOD is of variable size. Each SMOD contains the following information:

- Links to the AD and the previous SMOD
- Overlay information (for instance, its base address)
- Contents of A, B, C, E, and P registers

Additionally, an SMOD may contain the following information:

- Contents of operand registers
- Entries in the exit stack

The format of an SMOD is illustrated in the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

An SMOD is created when an overlay is first activated. When an overlay completes execution, either by returning control to the overlay that called it or by performing a GOTO service request to another overlay, its SMOD is deleted by removing all pointers to it. However, if the overlay performs a service request that results in the temporary loss of control (such as a CALL), the SMOD is updated with the executing environment of its overlay in anticipation of the return of control.

When an activity calls two or more overlays, multiple SMODs are retained. The collection of SMODs for a single activity is called a software stack. While the activity is executing, this software stack occupies an area of fixed size in Local Memory.

When an overlay performs a CALL, its SMOD is *pushed* onto the software stack. When the overlay regains control, its SMOD is *popped* from the stack. Each SMOD, except the SMOD for the first overlay of an activity, contains a link to the SMOD associated with the overlay that called it. For the first overlay, the pointer to the previous SMOD is 0.

When an activity relinquishes control of the IOP central processor to the Kernel and other activities are waiting on the central processor queue, the software stack for the activity losing control is written to Buffer Memory. The Local Memory software stack area is thus available to another activity. When the original activity regains control of the IOP central processor, its software stack returns to the software stack area in Local Memory. The AD remains in Local Memory and contains a link to the SMOD for the next overlay to be activated.

2.5 DEMON ACTIVITIES

Demon activities are created to perform tasks necessary to sustain I/O, but which take too much time and memory to execute directly in an interrupt handler.

Demon activities differ from normal activities in that they are assigned a Local Memory-resident SMOD. This provision minimizes activation time by eliminating the time normally needed for the SMOD exchange between Local and Buffer Memory.

The SMOD assigned to a Demon activity is large enough to contain the minimum amount of information needed to activate it. No room is provided for register saves or SMOD stacking, which means that Demon activities cannot perform Kernel service routines (such as CALL, PUSH, MSG, and RECEIVE) that could alter or temporarily suspend an activity.

Demon activities are created the same way as the normal activities, using the CREATE Kernel service routine. The DPTR keyword parameter on the CREATE macro (see the CREATE function later in this section) is used to specify that the activity being created is a Demon. The DPTR keyword is set to the Demon pointer D$*name*, which has been previously defined using the DAEMON macro in the Kernel Demons Table.

The D$*name* Demon pointer is used as an index into the Kernel Demons Table in the Kernel to find the AD for each Demon.

Example: Create ACOM Demon

Kernel Demons Table:

```
DEMONS      *
      .
      .
      .
ACOM        DAEMON
```

ACOM Demon creation:

```
EXT      D$ACOM
CREATE ACOM,,DPTR=D$ACOM,PRI=0
```

The preceding statement causes an AD and SMOD to be assigned and initialized for the ACOM Demon. The AD address is placed in the Kernel Demons Table, using D$ACOM as an index.

Demon activities are activated by making a call to the DACT Kernel subroutine. The D$*name* pointer assigned to the Demon to be activated is passed as a parameter in the accumulator.

Example: Activate ACOM Demon

```
EXT     D$ACOM              (Define if not in Kernel code)
A=D$ACOM
R=DACT
```

The D$*name* pointer is used as an index into the Kernel Demons Table to find the AD for the Demon. If the Demon is not already active or queued to be activated (AD@ACT=0), the AD address is passed to EQCP to be queued for activation.

2.6 OVERLAYS

Because of the limited size of Local Memory, the IOP software uses overlays. An overlay is an executable program or subroutine that normally resides in Buffer Memory. It is read into Local Memory when activated to perform some function.

The MIOP establishes the overlays in Buffer Memory and constructs the Overlay Table when the system is deadstarted. The Overlay Table contains an entry for each defined overlay. (The Overlay Table entry is illustrated in the IOS Table Descriptions Internal Reference Manual, publication SM-0007.) The entry for a particular overlay is derived from the overlay index, an equate of the form O$*ovlname*, defined in the overlay OVLNUM.

The Kernel uses the Overlay Table entry to load the associated overlay (if it is not already resident) when that overlay is called, and to determine the registers through which it receives parameters. If the overlay is called from a console, the parameters are the characters keyed in following the command, two characters per register; the maximum is one line of input. Internally called overlays are supplied parameters from the caller within the caller's registers.

During deadstart, the Kernel initializes the memory allocated for overlay space. The initialized space consists of three memory blocks: a header and a trailer delimiting the overlay space and one block containing the memory available for assignment. The amount of space allocated in a particular IOP is specified as a parameter in the overlay AMAP.

Two doubly linked lists run through the block headers. (See Local Memory Block Header in the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for a description of the header contents.) The two lists are as follows:

- The adjacent block list is ordered by block address and is used to calculate block sizes and to combine a block with adjacent free blocks when it is released.

- The memory search list links the free memory blocks and overlay blocks; the free blocks are kept at the head of the list and the overlays at the end, ordered in a least recently used manner.

When an overlay load is required, the Kernel scans the memory search list for an area large enough to accommodate the overlay. Overlay areas are released and combined with adjacent blocks, if necessary, as they are encountered during the scan. Upon finding a block of sufficient size, the Kernel creates a block for the overlay and reads the overlay into Local Memory. If the block is larger than that required, the Kernel also allocates a block of free memory, which is placed on the memory search list.

Whenever an overlay is entered (through a service function such as CALL, GOTO, RETURN, or CREATE), the memory block for that overlay is placed at the end of the memory search list.

The format of an overlay running under the Kernel is depicted in table 2-2. The first 4 parcels contain the name of the overlay, which has a maximum of 8 characters, zero-filled. All overlays are entered at parcel 6.

Overlays contain no variable data areas; that is, overlays are read-only programs and cannot be modified. An overlay obtains a data area by requesting a Local Memory area from the Kernel. Data areas thus allocated must be explicitly released when no longer needed. (The memory can be released in a different overlay or even in a different activity.)

Table 2-2. Overlay Format

Field	Parcel	Bits	Description
OV@NAM	0-3	0-15	Overlay name (up to 8 ASCII characters)
OV@TYP	4	0	Type of overlay: 0 Executable 1 Data
OV@NUM	4	1-15	Overlay number
OV@PAR	5	0-15	Parameter information:
SM@NUM	-	0-6	Number of registers
SM@FST	-	7-15	First operand register
OV@ENT	6	-	Entry point (first executable statement)

2.7 INTERRUPT PROCESSING

With the exception of the 100-Mbyte channels to Central Memory, SSD Memory, and Buffer Memory, IOS channels are normally managed through interrupts rather than by polling the Channel Done flag. That is, a function is initiated and control is relinquished to the Kernel, allowing the Kernel to do other useful work. A subsequent interrupt or time-out reactivates the software and processing continues.

When any peripheral function is completed (the done indicator is set), an interrupt is generated to the address contained in the first entry of the exit stack. That address is the same for any task running under the Kernel. All interrupts, therefore, go to the Kernel, which processes them in a manner consistent with the device causing the interrupt. Essential parts of interrupt handling routines are performed with interrupts disabled.

2.8 IOP CENTRAL PROCESSOR QUEUING AND ACTIVITY DISPATCHING

The Kernel maintains a queue containing the activities eligible for control of the IOP central processor. A simple priority scheme determines the order in which the activities receive control. The queue has 16 priority levels for activities, with priority 0 the highest and 15 the lowest. Demon activities performing Kernel functions run at a higher priority than other activities. A Kernel function can be interrupted by the hardware, but otherwise it gives up control only when its task is complete. The task is always of brief duration.

The priority of an activity is maintained within the Activity Descriptor and as such is semipermanent.

2.9 KERNEL SERVICE REQUESTS

The Kernel monitors the operation of IOP software and performs several service functions for software activities. These services are performed through well-defined interfaces between the activity and the Kernel.

2.9.1 GENERAL SERVICE FUNCTIONS

The general service functions are activity-oriented functions, including queuing and dequeuing operations, activity creation, and front-end communication.

2.9.2 MEMORY ALLOCATION AND DEALLOCATION

Each IOP has two types of Local Memory chains for allocating and deallocating memory for activities: the free memory chain and the fixed-size pool. The free memory chain is a block of memory that can be allocated in variable sizes. The first parcel of each allocated segment of memory is on a 4-parcel boundary. The Kernel always allocates in multiples of 4 parcels, regardless of the size of the request, by rounding up to the next multiple.

The other type of memory chain is the fixed-size pool, of which there are two types: the I/O buffer pool and the Disk Activity Link (DAL) pool.

Each memory piece in the I/O buffer pool is 2048 parcels in length and begins on a parcel boundary that is a multiple of 2048 parcels. This pool is used for storing data from disk sectors, on its way either to or from a disk attached to an IOP, and for other I/O device buffering.

Each memory piece in the DAL pool is 8 words (32 parcels) in length and begins on a 4-parcel boundary. This pool contains I/O requests from the mainframe and IOP-to-IOP messages while they reside in Local Memory.

The size of each of these pools is adjustable at assembly time; each processor has a pool size consistent with its function in the IOS. For example, the MIOP (the controller IOP) has few I/O buffers, but it has a relatively large quantity of free memory for use by overlays and station software. The Buffer I/O Processor (BIOP), because it largely moves data to and from Buffer Memory (and disk), assigns most of Local Memory to the I/O buffer pool.

Six Kernel requests allocate and deallocate memory from these pools. Two other requests allocate and deallocate Buffer Memory from the Kernel's pool of available memory.

2.9.3 I/O OPERATIONS

Activities executing in the IOS perform I/O for peripherals attached to the subsystem. In addition, the IOS software occasionally accepts files from front-end processors and writes them on disk at the direction of software executing in the mainframe. At other times, an IOP moves files from disk to a front-end computer for further processing.

The Kernel performs all of these functions. An activity running in an IOP makes a service request to the Kernel specifying the parameters for an I/O operation. The Kernel loads a special overlay containing the handler for the relevant device, and the function is performed while the activity waits for its completion.

Table 2-3 provides a brief summary of the service functions and the octal function codes. Each function is described in more detail in the following subsections, arranged alphabetically by function.

A macro call is the normal method for requesting service from the Kernel.

The address to which the service request macro performs its return jump is within the resident Kernel and is held in operand register %EX, which is dedicated to that function. The Kernel sets the register contents; the register must not be altered by any other software.

Following the Kernel call and the performance of the service function, control returns to the caller at the instruction after the return jump. After the call, the contents of the A and B registers depend on the type of call. The operand registers specified to be saved contain the values they contained at the time of the call, except in cases where they pass parameters back to the caller.

Table 2-3. Summary of Service Functions

Function Code	Name	Function
General Service Functions		
1	PUSH	Deactivates activity until popped
2	POP	Reactivates pushed activity
3	TERM	Terminates activity
4	GIVEUP	Reschedules activity
7	PAUSE	Suspends activity for specified tenths of a second
11	TPUSH	Pushes activity until popped or time expires
14	ASLEEP	Suspends activity until an AWAKE message is received
15	ALERT	Creates an activity in another IOP
16	AWAKE	Activates an activity in another IOP
17	RESPOND	Sends message response to an activity in another IOP
20	MSG	Sends message to Kernel console
21	MSGR	Sends message to Kernel console and waits for operator response
22	OUTPUT	Sends message to controlled CRT
24	A130OI	Performs front-end I/O on an NSC channel
25	RECEIVE	Waits for a character to be entered from a CRT controlled by USURP
37	OUTCALL	Calls (through the CALL function) an overlay in another IOP
50	CALL	Calls another overlay to perform a function

Table 2-3. Summary of Service Functions (continued)

Function Code	Name	Function
51	GOTO	Calls another overlay but does not save return information; returns to caller of this overlay.
52	RETURN	Returns to caller of the overlay; if none, terminates.
53	FIND	Returns the Buffer Memory address and length of an overlay
54	FLUSH	Releases all overlays in Local Memory
55	CREATE	Creates a new activity in the system
Memory Allocation and Deallocation		
26	GETDAL	Allocates local DAL
27	RELDAL	Releases local DAL
30	GETMEM	Allocates Local Memory in multiples of 4 parcels from free pool
31	RELMEM	Releases memory to free pool
32	BGET	Allocates a Local Memory I/O buffer of 2048 parcels
33	BRET	Deallocates a Local Memory I/O buffer
35	MGET	Gets Buffer Memory from pool of free buffers
36	MPUT	Returns Buffer Memory to free pool
Input/Output Operations		
34	SEND	Sends message to mainframe
42	HSPR	Reads data from a Target Memory into Local Memory

Table 2-3. Summary of Service Functions (continued)

Function Code	Name	Function
43	HSPW	Writes data from Local Memory to a target memory
44	POLL	Sends message to mainframe and waits for response
45	TRANSFER	Moves data between Buffer Memory and a target memory
46	MOSR	Reads data from Buffer Memory into Local Memory
47	MOSW	Writes data from Local Memory into Buffer Memory

2.9.4 FUNCTION DESCRIPTIONS

The following subsections describe the service functions in alphabetical order.

2.9.4.1 ALERT function (15)

ALERT creates an activity in a different IOP. Subsequent AWAKE functions can pass parameters to the newly created activity.

The stepflow for the ALERT service function (shown in figure 2-3) is as follows:

1. An activity in the originating IOP performs an ALERT service function, specifying the first overlay of the activity to be created and the IOP in which the activity will be created.

2. The Kernel in the originating IOP builds a message from the information provided in the service function and sends it to the Kernel in the target IOP by way of an interprocessor interrupt and Buffer Memory.

3. The Kernel in the originating IOP idles the activity that performed the service function.

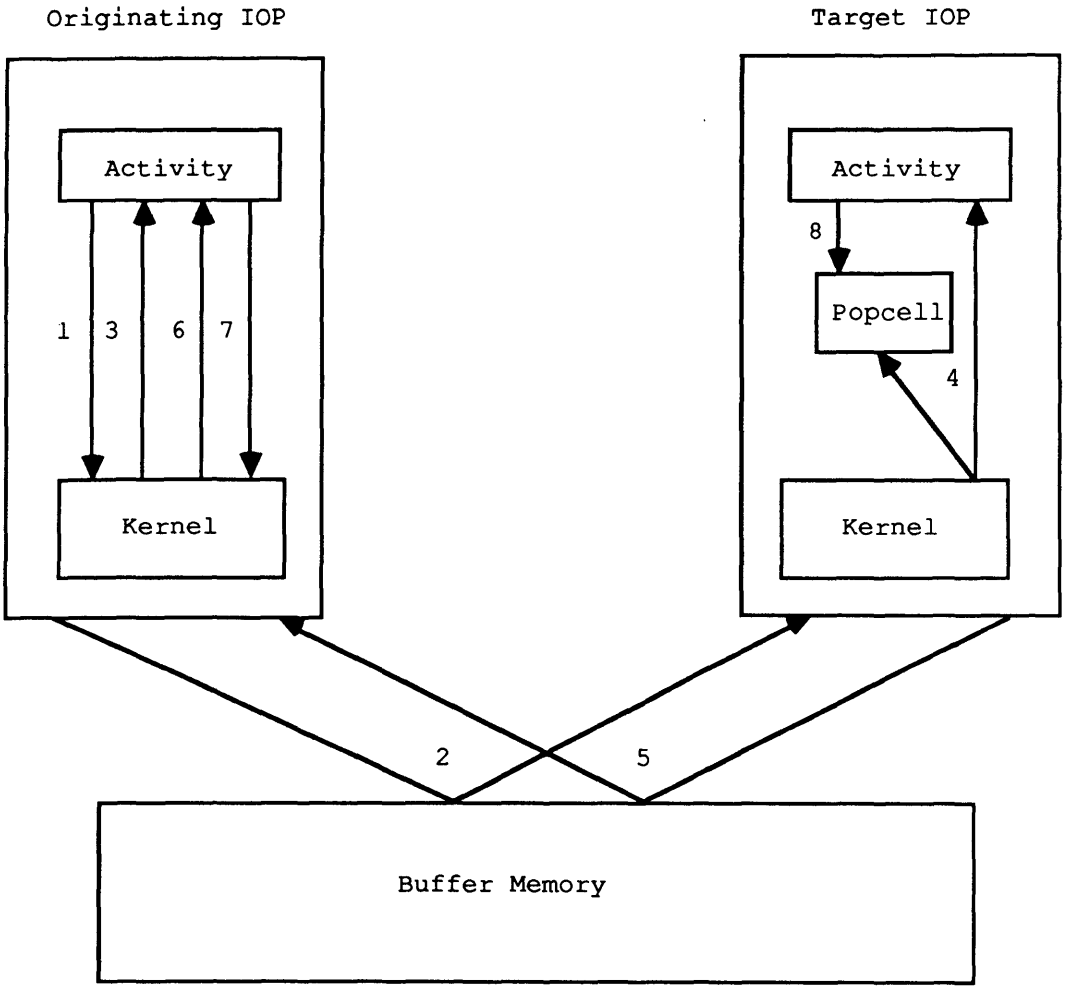


Figure 2-3. ALERT Stepflow

4. The Kernel in the target IOP performs the following operations:

   - Builds a *popcell* in Local Memory. A popcell contains linkage and message information allowing an activity in one IOP to communicate with an activity in another IOP.
   - Creates the new activity with a CREATE service function
   - Saves the popcell address; the address is supplied to the new activity in the first parameter register.
   - Places the Activity Descriptor on its central processor queue

5. The Kernel in the target IOP sends a message response containing the address of the popcell to the Kernel in the originating IOP by way of the Kernel's interprocessor message facility.

6. The Kernel in the originating IOP saves the popcell address.

7. The Kernel in the originating IOP places the originating activity on its central processor queue. When the activity is scheduled, the popcell address is supplied in the A register.

8. The newly created activity is popped off the central processor queue in the target IOP. The activity can issue ASLEEP functions to receive AWAKE requests generated in the originating IOP and RESPOND functions to respond to the requests.

Format:

Location	Result	Operand
	ALERT	*iop,overlay*

iop I/O Processor number:
- 0 IOP-0
- 1 IOP-1
- 2 IOP-2
- 3 IOP-3

overlay Name of initial overlay of activity to create

Example:

This example executes in MIOP and alerts overlay UCXFR (create a slave activity) in BIOP.

Location	Result	Operand
	ALERT	2,UCXFR

2.9.4.2 ASLEEP function (14)

ASLEEP gets messages supplied by AWAKE requests for activities created through ALERT functions.

If a message is queued on the popcell DAL queue, it is immediately returned to the activity. Otherwise, the activity is suspended until an appropriate AWAKE message is received.

All messages must be acknowledged through the RESPOND call so that resources allocated by the AWAKE request are released.

Format:

Location	Result	Operand
	ASLEEP	*popcell,dal*

popcell Popcell address supplied when the activity was created

dal Returned address of the message DAL

Example:

This example shows how a slave activity created by an ALERT request checks if any DALs, representing work to do, have been sent by the master activity.

Location	Result	Operand
	ASLEEP	R!P0,R!DAL

2.9.4.3 AWAKE function (16)

AWAKE passes parameters to an activity created earlier with an ALERT service function by sending it a Disk Activity Link (DAL). A DAL is a 32-parcel message. The Kernel uses the first 8 parcels for control purposes; parcels 8 through 31 may contain messages or data. The target activity receives messages by issuing the ASLEEP request.

The stepflow for the AWAKE service function (shown in figure 2-4) is as follows:

1. The originating activity builds a DAL with a message for the target activity and performs an AWAKE service function, specifying the target IOP, the popcell address, and the DAL address.

2. The Kernel in the originating IOP sends the DAL to the Kernel in the target IOP by way of Buffer Memory, using the interprocessor message facility.

3. The Kernel in the originating IOP idles the originating activity.

4. The Kernel in the target IOP places the DAL on the popcell DAL queue.

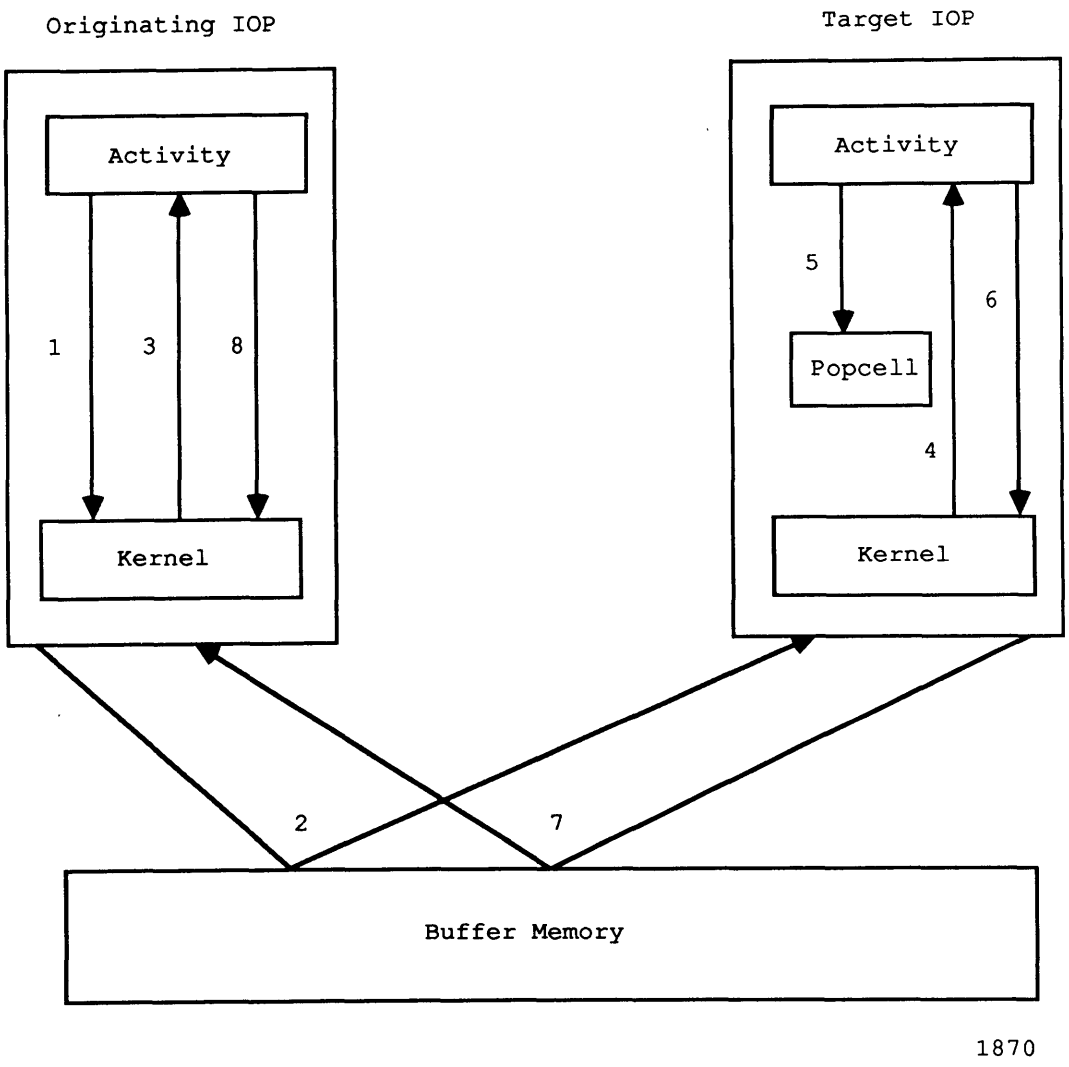


Figure 2-4. AWAKE Stepflow

5. The target activity processes the DAL returned by the ASLEEP request.

6. The target activity performs a RESPOND service function, and the Kernel in the target IOP places a status code in the message.

7. The Kernel in the target IOP sends the message to the Kernel in the originating IOP by way of Buffer Memory.

8. The Kernel in the originating IOP returns the status code to the originating activity in the A register and places the activity on its central processor queue.

Format:

Location	Result	Operand
	AWAKE	*iop,popcell,dal*

iop I/O Processor designator:
0 IOP-0
1 IOP-1
2 IOP-2
3 IOP-3

popcell Popcell address returned on the ALERT call

dal Message packet (DAL) address

Example:

This example executes in MIOP and shows how a master activity notifies the slave activity that there is work to do. The master created the slave by issuing an ALERT service request; then a DAL is sent to a specific slave in the BIOP.

Location	Result	Operand
	AWAKE	2,R!XFR,R!DAL

2.9.4.4 A130OI function (24)

The A130OI function performs as a read/write operation to a front end that is connected through an NSC A130 adapter. Control returns to the caller after the specified interrupt occurs.

Format:

Location	Result	Operand
	A130OI	*int,inl,in,chan,outl,out,q*

int Interrupt control; A '0' specifies an output channel interrupt, and a '1' specifies an input channel interrupt.

inl Input buffer length (in parcels)

in Input buffer starting address

chan Physical IOP input channel number

outl Output buffer length (in parcels)

out Output buffer starting address

q A 2-parcel queue address for Kernel queuing

Example:

This example shows one way to issue a read command to the A130 adapter. The constant value 1, signifies an input channel interrupt. The output buffer length is specified by using a table field name; NSB is the Input Status Buffer used to hold information relevant to the A130 adapter. LE represents the field name for the length of each entry in the table.

Location	Result	Operand
	A1300I	1,R!LEN,R!IN,R!CHAN,NSB@LE,R!OUT,R!Q

2.9.4.5 BGET function (32)

The BGET function allows an activity to get a fixed-size Local Memory buffer of 2048 parcels. The request is satisfied from the pool of I/O buffers. The first address of the buffer is a multiple of 512.

The error response, EC$BUFF, is returned in the A register if the function is unsuccessful. If successful, the address is returned in the specified register, and the A register is 0. The BGET call does not require a register save.

Format:

Location	Result	Operand
	BGET	*reg*

reg Operand register in which the buffer address is returned

Example:

This example shows that a 2-character register designator may be used to receive the address of the local I/O buffer.

Location	Result	Operand
	BGET	MA

2.9.4.6 BRET function (33)

The BRET function returns an I/O buffer of 2048 parcels to the pool of available buffers. The BRET call does not require a register save.

Format:

Location	Result	Operand
	BRET	*address*

address Address of buffer to be released

Example:

This example shows that a 2-character register designator may be used when making the service request. Like anywhere else is APML, the R! register notation could alternatively be used.

Location	Result	Operand
	BRET	MA

2.9.4.7 CALL function (50)

The CALL function activates an overlay to perform some service for the caller. The Kernel saves the caller's operand registers in a storage module, pushes the storage module on the software stack, loads the called overlay in Local Memory, and passes the parameters contained in the caller's operand registers. The number of parameters passed to the called overlay is determined by information the Kernel keeps about each overlay. The parameters are moved from the caller's registers to those of the called overlay.

After the called overlay completes the function for which it was called, it performs a RETURN function. The Kernel reloads the original caller (if necessary), loads its registers with the contents of the storage module, and returns control to the caller.

Format:

Location	Result	Operand
	CALL	*ovl*(,*pars*)[,TYPE=NUMBER]

ovl Name of the called overlay if TYPE=NUMBER is not specified; number of called overlay or register containing the called overlay number if TYPE=NUMBER is specified.

pars Parameters to be passed to the called overlay. Each register in which the caller expects a return parameter may be specified in the parameter list using the RO=*reg* option. The called overlay returns parameters to the caller through the RETREG macro described in section 14, Program Library and Macros. Blanks and extra commas are not allowed in the parameter list unless they serve as parameter space holders.

TYPE=NUMBER
If specified, *ovl* contains the overlay number or the register holding the overlay number.

Example:

This example shows a call to overlay BTO, by name. Parameters are passed to BTO according to the way the REGDEFS macro was used to define registers within overlay BTO. Notice that passed parameters may be registers, numeric constants, or symbols.

```
|Location |Result     |Operand
|         |           |
|         |CALL       |BTO,(R!MSGR,D'16,R!MSG,DTOFF,0)
```

2.9.4.8 CREATE function (55)

An activity can create an independent activity to run under the Kernel. The new activity is assigned a unique activity number distinguishing it from other activities. If the new activity is created successfully, the A register is 0 and the descriptor address of the new activity is returned in AD@P1 of the Activity Descriptor of the creator. If the creation is unsuccessful, the error code is returned in the A register.

The activity executing the CREATE function specifies the priority of the new activity, the initial overlay, and parameters for the new activity. The parameters are loaded into the overlay's specified registers when the new activity is activated.

The activity performing the CREATE function regains control immediately after the CREATE; the new activity is placed on the IOP central processor queue for later activation.

Format:

```
|Location |Result     |Operand
|         |           |
|         |CREATE     |ovl(,pars)[,PRI=pri][,DPTR=dptr][,TYPE=NUMBER]
```

ovl Name of the initial overlay in the created activity

pars Parameters to be passed to the created activity. Blanks and extra commas are not allowed in the parameter list unless they serve as parameter space holders.

PRI=*pri* Activity priority (0 through 15); the default is 8.

DPTR=*dptr* Index into Kernel Demons Table (D$*name*), if a Demon; default is AD$NODEM (normal activity, not a Demon).

TYPE=NUMBER
If specified, *ovl* contains the overlay number or the register holding the overlay number.

Examples:

This example shows the simplest form of CREATE. The name of the root overlay of the activity must be specified.

```
|Location |Result     |Operand
|         |           |
|         |CREATE     |CLKSNC
```

This example shows how to specify the overlay number of the root overlay of the activity. Notice that a priority may be specified in a register, and the demon index may be contained in a register. No parameters are being passed to the new activity.

```
|Location |Result     |Operand
|         |           |
|         |CREATE     |R!T0,,PRI=R!T1,DPTR=R!T2,TYPE=NUMBER
```

This example shows that passed parameters should be enclosed by parentheses. If more than one parameter is passed, refer to the CALL example to see that syntax. Priority of the CREATEd overlay can be specified with a numeric constant.

```
|Location |Result     |Operand
|         |           |
|         |CREATE     |UCSHL,(R!DAL),PRI=1
```

2.9.4.9 FIND function (53)

FIND returns the Buffer Memory address and length of an overlay in the registers designated. If the overlay does not exist, the error code EC$FIND is returned in the A register. If the overlay is found, the A register contains 0, and the specified operand registers contain the requested parameters upon return. The FIND call does not require a register save.

Format:

Location	Result	Operand
	FIND	*ovl,mosu,mosl,size*[,TYPE=NUMBER]

ovl Overlay name if TYPE=NUMBER is not specified; overlay number or register containing overlay number if TYPE=NUMBER.

mosu Operand register in which the high-order bits of the Buffer Memory address of the overlay are returned

mosl Operand register in which the low-order bits of the Buffer Memory address of the overlay are returned

size Operand register in which the word length of the overlay is returned

TYPE=NUMBER
If this parameter is specified, *ovl* contains the overlay number or the register holding the overlay number.

Example:

This example shows coding the FIND service request by using an overlay name and 2-character register designators to hold information returned by the Kernel.

Location	Result	Operand
	FIND	AMAP,T3,TA,T2

2.9.4.10 FLUSH function (54)

The FLUSH function releases all overlays in Local Memory. The memory occupied by the overlays is returned to the overlay space pool and is reallocated when overlays are read in from Buffer Memory.

Format:

```
|Location |Result     |Operand
|         |           |
|         |FLUSH      |
```

2.9.4.11 GETDAL function (26)

The GETDAL function allocates a DAL from the DAL pool. The address of the DAL allocated is returned to the register provided. If no DALs are available, an error response, EC$MEM, is returned to the caller's accumulator. The GETDAL function does not require a register save.

Format:

```
|Location |Result     |Operand
|         |           |
|         |GETDAL     |reg
```

reg Register to receive address of allocated DAL

Example:

This example shows how to code a GETDAL request.

```
|Location |Result     |Operand
|         |           |
|         |GETDAL     |R!DAL
```

2.9.4.12 GETMEM function (30)

The GETMEM function allocates Local Memory to the requesting activity from the free memory pool. All sizes are rounded upward to a multiple of four. The address returned is also a multiple of four. The requester is responsible for releasing the memory to the free pool when it is finished with it. If no memory is available, a zero address is returned to the requester and an error response, EC$MEM, is returned in the A register; the requester may then delay and repeat its request. If successful, the memory address is returned in the specified register, and the A register is 0. The GETMEM call does not require a register save.

Format:

Location	Result	Operand
	GETMEM	*size,reg[,tx0,tx1,tx2]*

size Number of parcels requested

reg Operand register in which the buffer address is returned

tx0,tx1,tx2
Optional ASCII name (maximum of 6 characters) stored in the 4-parcel header that identifies the owner of the buffer. If not supplied, the overlay name of the requester is used.

Examples:

This example shows a common form of the GETMEM request. Notice that the size is designated by a symbol.

Location	Result	Operand
	GETMEM	MEMSIZ,R!MSG

This example shows a less common form of coding a GETMEM request. A register is used to pass the size, and three parcels of text are being passed through registers.

Location	Result	Operand
	GETMEM	R!T3,R!T4,R!%W1,R!%W2,R!%W3

2.9.4.13 GIVEUP function (4)

The GIVEUP function allows an activity to relinquish control of the IOP in favor of higher priority tasks, if any exist. The Kernel places the activity on the IOP central processor queue at the activity's priority. If the current activity has the highest priority in the system, it regains control immediately.

Format:

Location	Result	Operand
	GIVEUP	

2.9.4.14 GOTO function (51)

The GOTO function activates an overlay without regaining control after the new overlay completes execution. The routine receiving control from the GOTO performs its function based on the parameters passed to it. When complete, it passes control to the caller of the routine that performed the GOTO. If such a routine does not exist in the software stack, the activity is terminated. The Kernel passes parameters in the operand registers specified in the B register of the caller. The number of parameters depends on the routine receiving the GOTO.

Format:

Location	Result	Operand
	GOTO	*ovl*(,*pars*)[,TYPE=NUMBER]

ovl Name of the called overlay if TYPE=NUMBER is not specified. Number of called overlay or register containing the called overlay number if TYPE=NUMBER is specified.

pars Parameters to be passed to the called overlay. Blanks and extra commas are not allowed in the parameter list unless they serve as parameter space holders.

TYPE=NUMBER
If specified, *ovl* contains the overlay number or the register holding the overlay number.

Examples:

This example shows how to pass control from one overlay directly to another. This example uses the name of the overlay to GOTO, and one parameter is passed by using a symbol.

Location	Result	Operand
	GOTO	CLINIT,(CLI$TERM)

This example shows how to use the overlay number on the GOTO statement. It also illustrates that many parameters may be passed to the new overlay.

Location	Result	Operand
	GOTO	R!%W2,(IA,IB,IC,ID,IE,IF,IG,IH,IJ,IK,IL,IM,IN, IO,IP,IQ,IR,IS,IT,IU,IV,IX,IY,IZ,R!IAA,R!IAB, R!IAC,R!IAD,R!IAE,R!IAF,R!IAG,R!IAH,R!IAI,R!IAJ, R!IAK,R!IAL,R!IAM,R!IAN),TYPE=NUMBER

2.9.4.15 HSPR function (42)

The HSPR function reads data from a target memory to Local Memory. If the request is made in an IOP with a 100-Mbyte channel connected to the target memory, the I/O is done immediately. If the request is made in an IOP that does not have a 100-Mbyte channel connected to the target memory, the following actions are performed:

1. The Kernel allocates a 512-word buffer in Buffer Memory.

2. A DAL is built that contains the target memory and Buffer Memory addresses and the number of words to read.

3. The DAL is sent to the Kernel in the IOP, which has a channel to the target memory, by way of Buffer Memory, using the interprocessor message facility.

4. The originating IOP's Kernel returns control to the calling activity if the NOWAIT parameter is specified or idles the activity if NOWAIT is not specified.

5. The AMSG activity in the Target Memory Processor reads the data from the target memory into the Buffer Memory buffer supplied by the originating IOP.

6. A response DAL is built and sent to the originating IOP, using the interprocessor message facility.

7. The originating IOP receives the response DAL and reads the requested data from Buffer Memory into a Local Memory buffer supplied on the HSPR call.

8. The Kernel deallocates the Buffer Memory buffer.

9. If NOWAIT was not specified, the status code returned from the Target Memory Processor is placed in the A register of the originating activity and the activity is placed on the central processor queue.

Format:

Location	Result	Operand
	HSPR	*tm,tmu,tml,buf,len*[,NOWAIT]

tm Target memory type:
FS$CMEM Central Memory
FS$SSD SSD Memory

tmu High-order bits of target memory address

tml Low-order bits of target memory address; must be on a 64-word boundary if target memory is FS$SSD.

buf Local Memory buffer address; must be on a word boundary.

len Length of transfer in 64-bit words; must be nonzero. Maximum length of transfer is 512 64-bit words, and length must be a multiple of 64 words if target memory is FS$SSD. The target memory address plus the length must not exceed the configured size of the target memory (CRAY@SIZ for Central Memory; SSD@SIZ for SSD Memory).

NOWAIT If NOWAIT is not specified, the I/O is completed before the activity is resumed. If NOWAIT is specified, the I/O is initiated and the activity is resumed immediately. The activity can then be made to wait for I/O completion, if a 100-Mbyte channel is present, by using a return jump to the Kernel subroutine CHNWTDN with B set to the 100-Mbyte input channel number. If a 100-Mbyte channel is not present, the calling activity should not use the NOWAIT option, because it cannot determine when I/O is complete.

Example:

This example shows how to code a "high-speed read" data transfer. The first parameter indicates whether the 100-Mbyte channel is connected to an SSD or Cray Central Memory.

```
|Location |Result     |Operand
|         |           |
|         |HSPR       |FS$CMEM,R!CU,R!CL,R!MPR,R!MOSL
```

2.9.4.16 HSPW function (43)

The HSPW function writes data from Local Memory to a target memory. If the request is made in an IOP with a 100-Mbyte channel connected to the target memory, the I/O is done immediately. If the request is made in an IOP that does not have a 100-Mbyte channel connected to the target memory, the following actions are performed:

1. The Kernel allocates a 512-word buffer in Buffer Memory.
2. The Kernel writes the data from the Local Memory buffer supplied on the HSPW call to the Buffer Memory buffer.
3. A DAL is built that contains the target memory and Buffer Memory addresses and the number of words to write.
4. The DAL is sent to the Target Memory Processor by using the interprocessor message facility.
5. The originating IOP's Kernel returns control to the calling activity if the NOWAIT parameter is specified or idles the activity if NOWAIT is not specified.
6. The AMSG activity in the destination IOP writes the data to the target memory from the Buffer Memory buffer supplied by the originating IOP.
7. A response DAL is built and sent to the originating IOP by using the interprocessor message facility.
8. The originating IOP receives the response DAL.
9. The Kernel deallocates the Buffer Memory buffer.
10. If NOWAIT was not specified, the status code returned from the Target Memory Processor is placed in the A register of the originating activity and the activity is placed on the central processor queue.

Format:

Location	Result	Operand
	HSPW	*tm,tmu,tml,buf,len*[,NOWAIT]

tm Target memory type:

FS$CMEM	Central Memory
FS$SSD	SSD Memory

tmu High-order bits of target memory address

tml Low-order bits of target memory address; must be on a 64-word boundary if target memory is FS$SSD.

buf Local Memory address; must be on a word boundary.

len Length of transfer in 64-bit words; must be nonzero. Maximum length of transfer is 512 64-bit words, and length must be a multiple of 64 words if target memory is FS$SSD. The target memory address plus the length must not exceed the configured size of the target memory (CRAY@SIZ for Central Memory; SSD@SIZ for SSD Memory).

NOWAIT If NOWAIT is not specified, the I/O is completed before the activity is resumed. If NOWAIT is specified, the I/O is initiated and the activity is resumed immediately. The activity can then be made to wait for I/O completion, if a 100-Mbyte channel is present, by using a return jump to the Kernel subroutine CHNWTDN with B set to the 100-Mbyte output channel number. If a 100-Mbyte channel is not present, the calling activity should not use the NOWAIT option, because it cannot determine when I/O is complete.

Example:

This example shows how to code a high-speed write data transfer.

```
|Location |Result     |Operand
|         |           |
|         |HSPW       |FS$CMEM,R!CU,R!CL,R!%W1,R!TLEN
```

2.9.4.17 MGET function (35)

The MGET function allows a user overlay to allocate one or more 512-word units of Buffer Memory for its own use.

When multiple buffers are requested, the allocated space consists of contiguous 512-word buffers. If the number of multiple buffers requested is not available, a smaller number is allocated.

Format:

```
|Location |Result     |Operand
|         |           |
|         |MGET       |upper,lower[,NUM=num][,GOT=got]
```

upper Operand register in which the high-order bits of the address of the first buffer are returned

lower Operand register in which the low-order bits of the address of the first buffer are returned

NUM=*num* Number of contiguous 512-word buffers to allocate. If *num* is not specified, 1 is assumed.

GOT=*got* Operand register in which the number of buffers allocated is returned; *got* is used when *num* is specified.

Example:

This example shows how to code an MGET request.

Location	Result	Operand
	MGET	R!MU,R!ML

2.9.4.18 MOSR function (46)

The MOSR function reads data from Buffer Memory to Local Memory.

Format:

Location	Result	Operand
	MOSR	*msu,msl,buf,len*[,NOWAIT]

msu High-order bits of Buffer Memory address

msl Low-order bits of Buffer Memory address

buf Local Memory address; must be on a word boundary.

len Length of transfer in 64-bit words; must be nonzero. Buffer Memory address plus length must not exceed the configured size of Buffer Memory (MOS@SIZ).

NOWAIT If NOWAIT is not specified, the I/O is completed before the activity is resumed; if NOWAIT is specified, the I/O is initiated, and the activity is resumed immediately. The activity can then be made to wait for I/O completion, if desired, by using a return jump to the Kernel subroutine CHNWTDN, with B set to the I/O channel number.

Example:

This example shows how to code a MOSR request.

```
|Location |Result     |Operand
|         |           |
|         |MOSR       |T3,T4,MA,T2
```

This example shows that numeric constants and symbols can be used as parameters.

```
|Location |Result     |Operand
|         |           |
|         |MOSR       |0,DEBSTART,R!NA,DEBLEN
```

2.9.4.19 MOSW function (47)

The MOSW function writes data to Buffer Memory from Local Memory.

Format:

```
|Location |Result     |Operand
|         |           |
|         |MOSW       |msu,msl,buf,len[,NOWAIT]
```

msu High-order bits of Buffer Memory address

msl Low-order bits of Buffer Memory address

buf Local Memory address; must be on a word boundary.

len Length of transfer in 64-bit words; must be nonzero. Buffer Memory address plus length must not exceed the configured size of Buffer Memory (MOS@SIZ).

NOWAIT If NOWAIT is not specified, the I/O is completed before the activity is resumed; if NOWAIT is specified, the I/O is initiated, and the activity is resumed immediately. The activity can then be made to wait for I/O completion, if desired, by using a return jump to the Kernel subroutine CHNWTDN, with B set to the I/O channel number.

Examples:

This example shows how to code a Buffer Memory write request. Numeric constants and symbols can be used as parameters.

```
|Location |Result     |Operand
|         |           |
|         |MOSW       |R!MU,R!ML,LBPT,1
```

This example shows the use of the optional NOWAIT parameter. It also shows that calculations will be performed as part of the macro processing of the request, so that R!%TIME>2 will be evaluated and passed as the actual parameter.

```
|Location |Result     |Operand
|         |           |
|         |MOSW       |0,R!%TIME>2,R!%TIME,2,NOWAIT
```

2.9.4.20 MPUT function (36)

The MPUT function request allows an overlay to return one or more 512-word units of Buffer Memory to the pool of free areas.

Multiple buffers returned must be contiguous.

Format:

```
|Location |Result     |Operand
|         |           |
|         |MPUT       |upper,lower[,NUM=num]
```

upper High-order bits of the address of the first buffer to release

lower Low-order bits of the address of the first buffer to release

NUM=*num* Number of contiguous 512-word buffers to release. If *num* is not specified, 1 is assumed.

Example:

This example shows how to code an MPUT request.

```
|Location |Result     |Operand
|         |           |
|         |MPUT       |R!MSU,R!MSL
```

2.9.4.21 MSG function (20)

An activity sends a message to the Kernel console with the MSG function. No response is expected. A formatted line is sent to the CRT, while the activity waits for the message to complete. The line must be in ASCII; binary zeros in the last character of the message signal the end of the line. The CRT-handling routine provides line feeds and carriage returns. The line is located in a Local Memory data area, outside of the overlay space. When the message is complete, the activity is placed on the IOP central processor queue for reactivation at its priority.

Format:

Location	Result	Operand
	MSG	*msg*

msg Message address

Example:

This example shows how to code an MSG request.

Location	Result	Operand
	MSG	R!MSG

2.9.4.22 MSGR function (21)

An activity uses the MSGR function to send a message to the Kernel console and to receive a response from the operator. A formatted ASCII line, with a binary zero for the last character, is sent to the console. The activity then waits until the operator enters a response to the message. The response is returned in a specified buffer and is also in ASCII.

After the response is entered (signified by a carriage return or line feed), the activity is placed on the IOP central processor queue for reactivation.

Format:

Location	Result	Operand
	MSGR	*msg,count,buffer*

msg Output message address

count Response buffer size in characters (allow one byte for line feed)

buffer Response message buffer address

Example:

This example shows that symbols may be used as parameters in the MSGR request.

```
|Location |Result    |Operand
|         |          |
|         |MSGR      |R!MSG,MSGRSIZ,R!MSGR
```

2.9.4.23 OUTCALL function (37)

The OUTCALL function calls an overlay in another IOP for execution. The Kernel passes the overlay number and a maximum of eight parameters to the destination IOP. The destination IOP creates an activity that calls the target overlay. When a RETURN function is executed, a response message is sent to the originating IOP, which reschedules the activity that issued the OUTCALL.

On return, the A register contains either the error code EC$IOP if the destination IOP is not configured or the response code returned by the called overlay, if applicable.

Format:

```
|Location |Result    |Operand
|         |          |
|         |OUTCALL   |iop,ovl(,pars)[,TYPE=NUMBER]
```

iop I/O Processor in which the called overlay is to execute

ovl If TYPE=NUMBER is specified, the number of the called overlay; otherwise, the name of the called overlay.

pars Parameters to pass to the called overlay (eight maximum)

TYPE=NUMBER
If specified, *ovl* contains the number of the called overlay.

Example:

This example shows how to OUTCALL an overlay in a different IOP. The name of the overlay is specified, along with the IOP in which the overlay should run. IOP may be designated with a symbol, or by a value (0 through 3) stored in a register. It is possible to pass parameters to the OUTCALLed overlay.

```
|Location |Result    |Operand
|         |          |
|         |OUTCALL   |MIOP,DKERR1,(R!%MYID,R!MU,R!ML)
```

2.9.4.24 OUTPUT function (22)

When an activity gains control of a CRT by calling the USURP overlay, the activity can send characters to the screen, one at a time, with the OUTPUT function. The Kernel neither interprets the output character nor provides any carriage return or line feed functions.

Format:

```
|Location |Result    |Operand
|         |          |
|         |OUTPUT    |device,base,offset,num
```

device CRT logical address (0 to 3)

base Output message parcel address

offset Output message byte offset from base

num Output message byte count

Example:

This example shows how to code an OUTPUT request.

```
|Location |Result    |Operand
|         |          |
|         |OUTPUT    |R!DEVICE,R!BUFFER,0,R!TEMP
```

2.9.4.25 PAUSE function (7)

The PAUSE function enables an activity to deactivate itself for a specified number of one-tenth second quanta. The activity is placed on a timer queue until expiration of the interval, when real-time clock interrupt code removes it from the queue and places it on the IOP central processor queue for activation at the priority level.

Format:

Location	Result	Operand
	PAUSE	*tenths*

tenths Interval before reactivation in tenths of a second

Examples:

This example shows that a common way to specify pause time is with a numeric constant.

Location	Result	Operand
	PAUSE	1

This example shows that it is possible to specify PAUSE time with a symbol.

Location	Result	Operand
	PAUSE	REFRESH

2.9.4.26 POLL function (44)

The POLL function allows an overlay to send a B or S type message packet to the mainframe and wait for a response. The message is a buffer of 6-words containing information recognized by COS. After sending the message, the activity is placed on a PUSH queue until a matching message returns from the mainframe. When the response returns, the waiting activity is removed from the queue and is reactivated to process the message received from the mainframe. Only overlays in the MIOP can make this request.

If the request is not answered within the time-out period, the message-sending activity is reactivated and status is returned so that it can perform error recovery.

Format:

Location	Result	Operand
	POLL	*message*

message Message packet (DAL) address

Example:

This example shows how to code a POLL request. The R!CXT in this example serves as a reminder that Cray packets/DALs are stored in CXT.

Location	Result	Operand
	POLL	R!CXT

2.9.4.27 POP function (2)

The POP function reactivates an activity that was placed on a queue with a PUSH function. The Kernel removes the activity from the top of the queue and places it on the IOP central processor queue at its priority for activation later. The message supplied is returned in the A register of the activity that was popped. A response code (either 0 or EC$EMPTY) is returned in the A register of the activity performing the POP. The POP call does not require a register save. When a pushed activity is subsequently popped, the A register contains a message from the routine that performed the POP.

Format:

Location	Result	Operand
	POP	*queue,message*

queue Queue address

message Message for popped activity

Examples:

This example shows that zero should be used as a parameter when there is no message to send to the popped activity.

```
|Location |Result     |Operand
|         |           |
|         |POP        |R!%W1,0
```

This example shows that a message can be passed through a register.

```
|Location |Result     |Operand
|         |           |
|         |POP        |R!MJ,R!MF
```

2.9.4.28 PUSH function (1)

The PUSH call deactivates an activity by placing the Activity Descriptor for the activity that is currently running on a specified queue. The Kernel then searches for other functions to perform. The AD is removed from the queue when another activity performs the POP function. The PUSH function is used when an activity needs a facility that is currently being used by another activity. When the other activity is finished with the facility, it activates the first by performing a POP. When a pushed activity is subsequently popped, the A register contains a message from the routine that performed the POP.

Format:

```
|Location |Result     |Operand
|         |           |
|         |PUSH       |queue[,order]
```

queue Queue address

order Order to add activity to queue. Default is to add activity by priority.
FIFO Add activity at queue tail
LIFO Add activity at queue head

Example:

This example show how to code a simple PUSH request.

```
|Location |Result     |Operand
|         |           |
|         |PUSH       |R!%W1
```

2.9.4.29 RECEIVE function (25)

An activity that has gained control of the keyboard of a CRT by calling the USURP overlay can perform a RECEIVE function. The activity is placed on the input queue of the designated CRT. The activity is dequeued and rescheduled when a character is received or when the specified time interval expires. If the A register contains 0 upon return, the input character is in the specified operand register; if A contains the EC$TIME message, the time limit was exceeded and no character is returned. The character is not echoed to the screen; that is the responsibility of the receiving activity.

Format:

Location	Result	Operand
	RECEIVE	*device,char,tenths*

device CRT logical address (0 to 3)

char Operand register to receive input character

tenths Maximum time to wait for input in tenths of a second

Example:

This example shows that the CRT address may be stored in a register, and wait time can be specified as a numeric constant.

Location	Result	Operand
	RECEIVE	R!DEVICE,R!CHAR,D'10

2.9.4.30 RELDAL function (27)

The RELDAL function returns a DAL to the DAL pool. The RELDAL call does not require a register save.

Format:

Location	Result	Operand
	RELDAL	*reg*

reg Register containing address of DAL to be returned

Example:

This example shows how to code the request to deallocate a Local Memory DAL.

Location	Result	Operand
	RELDAL	R!RESP

2.9.4.31 RELMEM function (31)

The RELMEM function deallocates a segment of memory previously allocated from the free memory pool. The requester supplies the address returned in a GETMEM call. The entire buffer allocated through the corresponding GETMEM call is released. The RELMEM call does not require a register save.

Format:

Location	Result	Operand
	RELMEM	*address*

address Address of buffer to be released

Example:

This example shows that it is common to store memory addresses in registers.

Location	Result	Operand
	RELMEM	R!MSG

2.9.4.32 RESPOND function (17)

The RESPOND function is performed by the activity in the IOP named in the AWAKE call after it processes the data in the request. A RESPOND function allows the Kernel to release the Buffer Memory message space and to reactivate the calling activity, if it is waiting.

Format:

Location	Result	Operand
	RESPOND	*dal,message*

dal Message packet (DAL) address

message Message returned in the A register of the activity doing the corresponding AWAKE call

Example:

This example shows how a slave activity ALERTed in one IOP will notify the master in a different IOP that a DAL has been processed.

```
|Location |Result     |Operand
|         |           |
|         |RESPOND    |R!DAL,R!UST
```

2.9.4.33 RETURN function (52)

The RETURN function in a CALL sequence indicates to the Kernel that the function is complete and control is returning to the previous entry in the software stack. The Kernel loads the registers of the previous entry from the storage module, adjusts its software stack pointers, reloads the overlay if necessary, and gives control to the caller of the overlay performing the RETURN function. The registers returned may contain parameters defined by the called routine. The RETURN call does not require a register save.

If no entries exist in the software stack, the Kernel terminates the activity.

Format:

```
|Location |Result     |Operand
|         |           |
|         |RETURN     |
```

2.9.4.34 SEND function (34)

The SEND function allows an overlay to send a 6-word message packet to the mainframe without waiting for a response. If the function successfully completes, the A register contains 0 upon return. If the low-speed channel to the mainframe is not enabled, the EC$NRDY error code is returned to the A register. Only overlays in the MIOP can make this request.

Format:

Location	Result	Operand
	SEND	*message*

message Message packet (DAL) address

Example:

This example shows how to code a SEND request.

Location	Result	Operand
	SEND	R!DAL

2.9.4.35 TERM function (3)

The TERM function signals the end of an activity's processing. The Kernel terminates the activity and releases its Activity Descriptor, Buffer Memory software stack, and possibly the popcell. The TERM call does not require a register save.

Format:

Location	Result	Operand
	TERM	

2.9.4.36 TPUSH function (11)

The TPUSH macro deactivates an activity until either the time interval expires or a POP is performed by some other activity. Either occurrence places the activity on the IOP central processor queue for activation. When an activity executing the TPUSH macro is eventually popped, the A register contains either a message from the activity performing the POP or the response code EC$TIME if the time limit expired.

Format:

Location	Result	Operand
	TPUSH	*queue,tenths*

queue PUSH queue address

tenths Interval before activation, if not popped, in tenths of a second

Examples:

This example shows how to use registers in a TPUSH request.

```
|Location |Result    |Operand
|         |          |
|         |TPUSH     |R!%W1,R!%W2
```

This example shows that calculations will be performed before the actual argument is sent to the Kernel, and time can be specified with a symbol.

```
|Location |Result    |Operand
|         |          |
|         |TPUSH     |R!Q+1,CT$RW
```

2.9.4.37 TRANSFER function (45)

The TRANSFER function moves data between Buffer Memory and a target memory. If the request is made in an IOP that has a 100-Mbyte channel connected to the target memory, the I/O is done immediately. If the request is made in an IOP that does not have a 100-Mbyte channel connected to the target memory, the following actions are performed:

1. A DAL is built that contains the target memory and Buffer Memory addresses, the direction of transfer, and the number of words to transfer.

2. The DAL is sent to the Target Memory Processor using the interprocessor message facility.

3. The originating IOP's Kernel returns control to the calling activity if the NOWAIT parameter is specified or idles the activity if NOWAIT is not specified.

4. The AMSG activity in the Target Memory Processor moves the data between the Target Memory and Buffer Memory in the direction requested.

5. A response DAL is built and sent to the originating IOP using the interprocessor message facility.

6. The originating IOP receives the response DAL.

7. If NOWAIT was specified, the status code returned from the Target Memory Processor is placed in the A register of the originating activity and the activity is placed on the central processor queue.

Format:

Location	Result	Operand
	TRANSFER	*dir,tm,tmu,tml,msu,msl,len*[,NOWAIT]

dir Direction of transfer:
- FS$IN From Target Memory to Buffer Memory
- FS$OUT From Buffer Memory to Target Memory

tm Target memory type:
- FS$CMEM Central Memory
- FS$SSD SSD Memory
- FS$BMR BMR Memory

tmu High-order bits of Target memory address

tml Low-order bits of target memory address; must be on a 64-word boundary if target memory is FS$SSD.

msu High-order bits of Buffer Memory address

msl Low-order bits of Buffer Memory address

len Length of transfer in 64-bit words; must be nonzero. Maximum length of transfer is 65,535 64-bit words, and length must be a multiple of 64 words if target memory is FS$SSD. The target memory address plus the length must not exceed the configured size of the Target Memory (CRAY@SIZ for Central Memory; SSD@SIZ for SSD Memory; BMR@SIZ for BMR Memory). The Buffer Memory address plus the length must not exceed the configured size of Buffer Memory (MOS@SIZ).

NOWAIT If NOWAIT is not specified, the I/O is completed before the activity is resumed. If NOWAIT is specified, the I/O is initiated and the activity is resumed immediately. The activity can then be made to wait for I/O completion, if a 100-Mbyte channel is present, by using a return jump to the Kernel subroutine CHNWTDN with B set to the 100-Mbyte I/O channel number. If a 100-Mbyte channel is not present, the calling activity should not use the NOWAIT option, because it cannot determine when I/O is complete.

Example:

This example shows how a TRANSFER request may be coded. Note that direction of transfer and target memory had previously been loaded into registers R!SC2 and R!TM, respectively.

```
|Location |Result     |Operand
|         |           |
|         |TRANSFER   |R!SC2,R!TM,R!CPU,R!CPL,R!SC0,R!SC1,R!LN1
```

2.10 CLOCK FUNCTIONS

The IOS real-time clock provides a system interrupt once every millisecond. This fixed time interrupt allows the operating system to time out events (such as pending interrupts) as well as maintain the time of day.

2.10.1 REAL-TIME CLOCK INTERRUPT HANDLER

The real-time clock interrupt handler is given control when a real-time clock interrupt occurs. Its functions are to do the following:

- Clear the interrupt
- Increment the interval counter (%MSEC) by 1. When the interval counter (%MSEC) reaches 100, indicating a one-tenth second interval, the clock demon (CLOCK) is activated and the counter (%MSEC) is reset to 0.

2.10.2 CLOCK DEMON

The clock demon (CLOCK) is activated once every tenth of a second by the real-time clock interrupt handler. Its functions are as follows:

- Services the event timer. Decrement 1 from TMR@TM of each entry linked to the timer queue (RTCQUE). If the decrement results in a count of 0, a time-out is indicated and the following occurs:
  - The entry is unlinked from the timer queue. The TMR@RT field in the entry is checked for nonzero; if nonzero, a return jump to that address occurs, and the entry address in register R!EH is passed.

- If TMR@RT is 0, the entry is assumed to be AD@P1 of an Activity Descriptor, and the following occurs:

  The function code that caused the entry to be placed on the timer queue is checked for a POLL (AD@FU=F$POLL). If the function was a POLL, the DAL that was being polled is removed from the MIOP-mainframe poll queue (CPI@PQ).

  If the function was not a poll, AD@RC is checked for an event queue address. If it is nonzero, the activity is removed from the event queue.

- Finally, the time-out code (EC$TIME) is placed in the activity return code (AD@RC) and the activity is reactivated.

- Increments the 1-second interval counter (%TENTHS). If the counter has not reached 10, indicating a 1-second interval, CLOCK terminates.

- Resets the 1-second counter to 0.

- Updates the idle time, Buffer Memory channel transfers, and 100-Mbyte channel transfers for the last second.

- Checks for MIOP-mainframe output channel time-out (CPO@TO#O). If the condition is found, activate the NOBEAT activity to display a message on the Kernel console.

- If 1 minute has expired (SECOND=0) in MIOP, sends a heart beat signal (M$SYNCH) to each configured IOP. Check to see if all IOPs signaled on the previous minute boundary have responded. If any were found to have not responded, activate the NOBEAT activity to display a message on the Kernel console.

- Updates the day clock in MIOP.

- If not running in MIOP, deselects inactive disks.

2.10.3 SYSTEM EVENT TIMER

The system event timer provides the means to regain control when an expected event does not occur within the expected time. It also allows an activity to give up control for a specified amount of time (see the PAUSE function earlier in this section).

The system event timer consists of the following elements:

Element	Description
QTIME	A Kernel subroutine called to link entries to the timer queue (RTCQUE). The following parameters are passed: • R!EH - Entry address • R!EG - Time Quantum in tenths of a second
DQTIME	A Kernel subroutine called to remove an entry from the timer queue. The following parameter is passed: • R!EH - Entry address
CLOCK DEMON	Activated every tenth of a second by the real-time clock interrupt handler. It is responsible for decrementing the time-out count for each entry (TMR@TM) and processing any time-outs that occur (TMR@TM=0).
TIMER ENTRY	Any system routine wishing to use the event timer is responsible for the allocation and maintenance of its timer entry, including setting a time-out routine address in TMR@RT, calling QTIME to link the entry, and calling DQTIME to unlink the entry, if the event being timed occurs before the timer expires. Activities using the service requests for timing events (TPUSH, PAUSE, and so on) do not need to know about the timer mechanisms. The timer entry for each activity is contained in its Activity Descriptor (AD@P1) and all maintenance is handled by the Kernel.

2.11 IOP DEADSTART

The original deadstart of the IOS occurs from either the Peripheral Expander tape or Peripheral Expander disk. The Deadstart package is read directly into MIOP Local Memory, beginning at address 0 and continuing until complete.

Once this operation is complete, the MIOP can read and write from the channels attached to it. The MIOP initializes Buffer Memory and any common tables residing there.

Once Buffer Memory is established and the MIOP is running, the MIOP deadstarts the other IOPs in the configuration. This is accomplished by sending a special function across the accumulator channel that reads Buffer Memory into Local Memory. Once read, the special function starts the IOP at address 0. Processors started in this way have access to tables and data in Buffer Memory that they can use to initialize their Local Memory.

2.12 STATISTICS

The Kernel keeps statistics on important events within the operating environment. The statistics consist mainly of counts of the occurrence of various phenomena and may be useful in tuning the system to maximize throughput and efficiency. The following occurrences and events are monitored:

- Buffer Memory references, including both input and output
- Disk channel references and the number of times error recovery is called, by channel
- Number of communications among IOPs
- Number of mainframe channel interrupts
- Number of 100-Mbyte channel transfers to or from Central Memory
- Number of 100-Mbyte channel transfers to or from SSD Memory

2.13 COMMUNICATION AMONG IOPs

Communication among IOPs occurs across accumulator channels. One parcel (16 bits) of information can be passed through the accumulator through the interrupt mechanism.

The message can be either in the accumulator itself or in a portion of Buffer Memory specified in the accumulator. In the latter case, an address within the Buffer Memory communications area of the sending IOP and a function code telling what sort of action is being requested are specified in the accumulator. Some messages can require a packet of information in the Buffer Memory data area specified in the accumulator. The data area indicated is an offset from the beginning of the communications area assigned to the sending IOP.

For a listing of the function codes and their meanings, see table 2-4.

Table 2-4. I/O Processor Intercommunication Function Codes

Function Code	Definition
0	The command code, contained in bits 4 through 15, is one of the following: M$GO Initiate SYSDUMP processing M$SYNCH Synchronize IOP software clock No Buffer Memory data is associated with these codes.
1-3	Unused
4	The message is contained in the Buffer Memory message area of the MIOP at an address specified in the low-order 12 bits of the accumulator. Each message area consists of eight 64-bit words. To find the Buffer Memory address, shift the accumulator left 3 bit positions and add the base address of the Buffer Memory message area for the MIOP. The message is intended for the BCOM overlay.
5	The message is in the area controlled by the BIOP for messages to the other processors. Otherwise, the protocol is the same as for function code 4. The message is routed to BCOM.
6	The message is in IOP-2 message area and is routed to BCOM.
7	The message is in IOP-3 message area and is routed to BCOM.
10	The message is in the MIOP message area and is routed to ACOM.
11	The message is in the BIOP message area and is routed to ACOM.
12	The message is in IOP-2 message area and is routed to ACOM.
13	The message is in IOP-3 message area and is routed to ACOM.

Table 2-4. I/O Processor Intercommunication Function Codes (continued)

Function Code	Definition
14	The message is in the IOP message area indicated in bits 4 and 5 of the accumulator. The low-order 10 bits specify the Buffer Memory address. The message is intended for the ICOM overlay.
15	The message is in the IOP message area indicated in bits 4 and 5 of the accumulator. The low-order 10 bits specify the Buffer Memory address. The message is intended for the HCOM overlay.

The sender of the message controls allocation and deallocation of message areas within its own Buffer Memory space. The IOP receiving a message informs the sending IOP when the function is complete and the message area can be deallocated.

An accumulator message of the form 177*nnn*; *nnn* is an error code, indicates a fatal error in the sending IOP and requests that the receiving IOP terminate normal operations. Otherwise, the accumulator contains the octal function code in its first 4 bits and the address or command in its final 12 bits.

2.14 MIOP-MAINFRAME COMMUNICATION CHANNEL

All communication between the IOS and the mainframe is in the form of fixed-length packets passed back and forth across a pair of 6-Mbyte asynchronous channels. The communication packets are referred to as DALs in the IOS.

A DAL is composed of two parts: the header (DA@@LH), which contains information used internally by the IOS, and the entry (DA@@LE), which is the actual information exchanged with the mainframe.

The first 2 parcels of the DAL entry are standard for all packets exchanged between the mainframe and the IOS. These parcels contain the source and destination information used for routing packets to appropriate routines.

2.14.1 MIOP-MAINFRAME COMMUNICATION INITIALIZATION

Before communication can begin between the IOS and the mainframe, a handshaking sequence must occur to ensure that the mainframe and IOS are synchronized. The CRAY overlay is called either by the deadstart process or by operator command to accomplish the following:

1. Clear both the input and output channels
2. Strip any data currently on the input channel
3. Poll an RQ$INIT0(I) packet; validate content returned.
4. Poll an RQ$INIT1(J) packet; validate content returned.
5. Set input/output channels enabled (CPO@ON/CPI@ON)

2.14.2 INPUT CHANNEL FROM THE MAINFRAME

The normal state of the input channel from the mainframe is busy and not-done, which means the channel is open to accept a packet from the mainframe at any time. (It is open to the entry portion of a DAL.) When the mainframe sends a packet to the IOP, the input channel state (done and not-busy) generates an interrupt in the MIOP.

The input channel has a table associated with it (CPI@), which is described in the IOS Table Descriptions Internal Reference Manual, publication SM-0007. The address of the table is kept in a global register (%LSPI).

The input interrupt handler has the following functions:

- Validates and saves the channel status (CPI@ST)
- Validates and saves the ending channel address (CPI@CA)
- Determines by destination ID (DA@DID) who to send the packet to
- Reopens the channel to another DAL for the next message. If no DALs are available, the channel is disabled (CPI@ON=0).

2.14.3 INPUT PACKET DISPOSITION

All input packets are checked for a recognized destination ID (DA@DID). Based on this ID, the packets are dealt with in three different ways, as follows:

- Packets with a destination ID of RQ$STAT (station) or RQ$PERF (statistics) can only be received in response to a poll. These packets have a code (CXCNT) which is matched to a DAL on the poll queue (CPI@PQ). The activity pointed to by the matched DAL (DA@ACT) is reactivated with the address of the DAL just received (DA@HP0).

- A packet with a destination ID (DA@DID) of RQ$KERN (Kernel request) is checked for a code of KF$KILL in CXKFC of the packet. This is a mainframe request for the IOS to crash. The IOS obliges immediately.

- Packets with a destination ID of RQ$DISK, RQ$HSX, or RQ$BMX0 are placed on the CDEM demon queue (CPI@CQ) for disposition to other IOPs or special processing in the MIOP. Packets with a destination ID of RQ$UCHN, RQ$KERN, or RQ$TTY are placed on the ADEM demon queue (DPI@AQ) for processing.

2.14.4 OUTPUT CHANNEL TO THE MAINFRAME

The output channel to the mainframe sends messages from the IOS to the mainframe. When a message is to be sent to the mainframe, a call is made to the SEND Kernel service request or IDALSND routine with the address of the DAL to send.

If the channel is busy, the DAL is placed on a queue in the Output Channel Table (CPO@QU); otherwise, the channel is immediately opened to the entry portion of the passed DAL. When the mainframe accepts the message from the channel, an interrupt is generated in the MIOP. The output channel has a table associated with it (CPO@). The address of this table is kept in a global register (%LSPO).

The output interrupt handler has the following functions:

- Clears the interrupt

- Validates and saves the channel status (CPO@ST)

- Validates and saves the ending channel address (CPO@CA)

- Checks the queue (CPO@QU) for more messages to send. If there are more messages, a call is made to IDALSND to transfer the next message on the queue.

- Checks the state of the input channel (CPI@ON). If off, the input channel is reenabled with the DAL that contained the message just accepted by the mainframe.

2.15 ERROR PROCESSING

IOS error processing is handled in two different methods, depending on the serial number of the IOS. IOSs with serial numbers of 21 or less use an error channel on the MIOP for error processing. IOSs with serial numbers greater than 21 use an error multiplex that passes error information to a maintenance computer. The following subsections describe these two methods.

2.15.1 ERROR CHANNEL PROCESSING (IOS SERIAL NO. 21 AND BELOW)

The IOS error channel exists only in the MIOP. Subroutines for processing interrupts on this channel reside only in the MIOP. (These subroutines are overwritten in the other IOPs and the space used for other purposes.)

An interrupt on the error channel (channel 16) indicates an error in Local Memory of one of the other IOPs, in Buffer Memory, in Central Memory, or on the 100-Mbyte channel. Local Memory errors in the MIOP are reported on the Local Memory error channel in the MIOP (channel 3).

When interrupts occur on the error channel, the hardware retains 4 parcels of information, which the software can access through registers to learn the type and location of the error. (A complete description of the error information can be found in the IOS hardware reference manual for your system). Four registers contain the following error information:

Register	Contents
1	The interface error status register contains a bit for each possible error. If the bit is set, the corresponding error has occurred. The error status register bits are as follows:

Bit	Control Signal
2^0	BIOP, Local Memory error
2^1	IOP-2, Local Memory error
2^2	IOP-3, Local Memory error
2^3	Buffer Memory error
2^4	Central Memory error
2^5	100-Mbyte channel input A error
2^6	100-Mbyte channel output B error
2^7	100-Mbyte channel input C error
2^8	100-Mbyte channel output D error

2	Parameter 1, containing special information that depends on the error type
3	Parameter 2, containing the low-order bits of the address of a Central Memory or Buffer Memory error. This parameter is not meaningful on Local Memory errors.
4	Parameter 3, containing the high-order bits of the address of a Central Memory or Buffer Memory error. This parameter is not meaningful on Local Memory errors.

2.15.1.1 Interrupt answering

When an interrupt occurs on the error channel, interrupt answering handles the possible errors singly. Interrupt answering reads the error status register and begins processing from the rightmost bit (for the BIOP Local Memory errors).

If a bit is set, interrupt answering reads the parameter registers for that type, puts the status into the Error Log Table (ERRLOG) in the Kernel, and counts the error. (See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for the format of the Error Log Table.)

Interrupt answering builds an error log packet (type C) containing the 4 status parcels (parcels 4 through 7 in the packet) and sends it across the 6-Mbyte channel to be processed by the mainframe.

The MIOP maintains a table in Buffer Memory containing the last 512 errors reported on the channel. The table is circular and has 4 parcels of information about each error (the contents of the parameter registers). The information may be printed through an ERRDMP command at the Kernel console. (See the IOS operator's guides for the ERRDMP Kernel command).

Interrupt answering processes the errors that were indicated in the original status register readout. When all errors are processed and logged, interrupt answering returns to routine ICHK to check for interrupts on other channels.

If more than 65,535 errors are reported to the error channel, the software automatically turns off the error channel so that no more interrupts are taken. The channel can be turned on with an ERROR ON command at the Kernel console (see the IOS operator's guides). Local Memory errors in the MIOP are processed and sent to the mainframe in a similar manner, except that Local Memory errors in the MIOP cause a Kernel halt. Therefore, the information is not written to the circular table in Buffer Memory.

2.15.1.2 Retrieving error log information

Information about errors can be obtained from the COS system log by using EXTRACT, from UNICOS using **errpt**, or from one of the following IOS-resident operations:

Operation	Description
ERRDMP	Entered at the Kernel console, ERRDMP prints the contents of the circular table in Buffer Memory containing the last 512 errors.
ERROR	This station command displays error status information as specified in the IOS operator's guides.

2.15.2 ERROR LOGGING (IOS SERIAL NO. 21 AND UP)

IOSs with serial numbers greater than 21 use an error multiplex for detecting and reporting IOS errors. This multiplex passes channel error information and memory error information to a maintenance computer. The maintenance computer program logs the error information for later analysis.

The multiplex module captures single and multiple errors, even if the multiple error is embedded in a burst of single-bit errors. Channels that are multiplexed include the Buffer Memory channels, the Local Memory channels, and the 100-Mbyte channel pairs to Central Memory.

3. DISK INPUT/OUTPUT

The I/O Subsystem (IOS) provides control for disk input and output. This section describes the components of the disk-controlling software in the following order:

- DCU-4 controlling software
- DCU-4 disk error recovery
- DCU-5 controlling software
- DCU-5 disk error recovery
- Striped disk groups
- Kernel internal disk I/O

The IOS disk software performs disk I/O and error recovery for the mainframe and for routines internal to the IOS.

The IOS can access a maximum of 48 disk storage units (DSUs). The maximum is attained by a system with four I/O Processors (IOPs), three of which have the maximum of 16 DSUs attached. All of the DSUs can be selected concurrently, but the number of data streams that can be maintained is limited. This limit is based on the disk device type, the number of 100-Mbyte channels configured, and the amount of Local Memory and Buffer Memory available for disk use.

Two independent disk drivers support six types of DSUs on the IOS. One driver supports the DD-19 and DD-29 DSUs through the DCU-4 Disk Controller. The other driver supports the DD-39 Disk Unit, DS-40 Disk Subsystem, RD-10 Disk Subsystem, and DD-49 Disk Unit through the DCU-5 Disk Controller. Both drivers may execute in the same IOP simultaneously, allowing all types of disks to be configured on the same IOP.

Current Cray disk conventions allocate space on disk at installation time for diagnostic system files and scratch areas. See the COS Operational Procedures Reference Manual, publication SM-0043, or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018 for specific information on reserved areas.

3.1 REQUEST PROCESS OVERVIEW

The mainframe initiates disk I/O by sending information to the IOS in a request packet. The information sent includes a starting device address, a target memory type and starting address, a transfer word length, and a read/write function code. The target memory may be Central Memory, SSD Memory, or the Buffer Memory resident (BMR) dataset portion of Buffer Memory. See the COS Operational Procedures Reference Manual, publication SM-0043, or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018, for more information on BMR datasets.

The IOS disk software validates the request parameters and sends an error status back to the mainframe if any illegal values are detected. I/O is then performed as efficiently as possible using Local Memory disk buffers, the 100-Mbyte channel connected to the specified target memory, and the optional Buffer Memory disk buffers. All disk buffers are 512 decimal words in length. When I/O is complete, the IOS sends the status back to the mainframe in the original request packet.

Buffer Memory disk buffers are used by the disk software in data caching mechanisms called "Read-ahead" and "Write-behind." These mechanisms are designed to facilitate data streaming by overlapping the disk I/O with request preparation in the mainframe.

3.2 DCU-4 CONTROLLING SOFTWARE

This subsection describes the architecture and request process for the DCU-4 (DD-19 and DD-29) controlling software.

3.2.1 DCU-4 SOFTWARE OVERLAYS

A set of overlays, in cooperation with disk interrupt answering, performs the actions necessary to stream data to DCU-4 disk devices. Each overlay is activated by the Kernel in the normal overlay activation process. The overlays ACOM, CDEM, and DISK are, however, demon processes and thus have only minimal SMODs associated with them. Therefore, parameters are not passed to them through registers, and contents of registers are not preserved if they perform Kernel service requests.

ERRECK, the error processor, runs as a normal overlay. Its activity is created through a service request by DISK. Each time ERRECK is created, it handles errors only for the channel for which it was created.

The following subsections describe the overlays and resident subroutines that handle the streaming of data to DCU-4 disk devices.

3.2.1.1 ACOM overlay

ACOM handles messages from other IOPs and initiates disk processing of new requests. It runs in all IOPs.

ACOM's message function codes are as follows:

Code	Function
1	Initiate disk request
2	Release Disk Activity Link (DAL) in originating IOP
3	Transfer data from target memory to Buffer Memory
4	Transfer data from Buffer Memory to target memory
5	Send status to mainframe; the MIOP receives this function.
6	Target memory to Buffer Memory transfer done
7	Buffer Memory to target memory transfer done

3.2.1.2 CDEM overlay

CDEM is the MIOP overlay that dispatches requests from the mainframe to the correct recipient. Messages processed by CDEM include disk, tape, and Kernel requests.

3.2.1.3 DISK overlay

DISK runs in all IOPs that have attached disks. It performs the following functions:

- Manages the Disk Control Block (DCB) done queue
- Initiates requests to the target memory processor to receive data from or send data to the target memory
- When executing in the target memory processor, moves data to the target memory through the 100-Mbyte channel
- Starts I/O if the channel is disabled (the channel can be disabled while waiting for data or a local disk buffer)

3.2.1.4 ERRECK overlay

ERRECK processes errors, attempts to recover from errors, and puts out the unrecoverable error message. It executes in all IOPs with attached disks.

3.2.1.5 Disk interrupt answering subroutine

The disk interrupt answering subroutine puts the finishing DAL on the DCB done queue, allocates local buffers on reads, and initiates I/O for the next DAL. Finally, it activates the DISK demon to further process the I/O completed on the channel.

3.2.1.6 Disk driving subroutines

Disk driving subroutines are called by ACOM and DISK to build executable DALs. These subroutines calculate the cylinder, head, sector, and the target memory address of data. They also allocate Buffer Memory space and initiate I/O on read requests, if necessary.

3.2.2 DCU-4 TABLES AND PACKET STRUCTURE

When the mainframe initiates communication with the IOS, it sends a 6-word (30_8-parcel) packet to MIOP across the 6-Mbyte channel with information about the request. Two words (10_8 parcels) of control information are added to the packet, which is called a master DAL. The contents of that packet are defined in the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

3.2.3 STEPFLOW FOR DCU-4 DISK WRITE REQUEST FROM MAINFRAME

The following sequence of operations handles a write to a disk attached to the DIOP. The target memory processor is BIOP. The IOP in which each step occurs is identified.

Step	IOP	Description
1.	MIOP	An interrupt on the 6 Mbyte channel is recognized as a disk request by interrupt answering. The CDEM overlay is activated.
2.	MIOP	CDEM forms a DAL using 30_8 parcels of information from the mainframe and 10_8 parcels of control information. The DAL function code is set to 1 (in DA@IFC) and put on the channel queue that connects IOPs, targeted for the

Step	IOP	Description
		DIOP. Buffer Memory is allocated and the address is saved in the DAL. DA@MES contains the encoded address in Buffer Memory of the message. (See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for an example of a DAL format.)
3.	MIOP	A message crosses the accumulator channel to the DIOP.
4.	DIOP	The interrupt handler gets the accumulator channel data and puts it on the ACOM activities data queue. ACOM is activated by being placed on the IOP central processor queue.
5.	DIOP	ACOM gets the entry from the queue, calculates the Buffer Memory address of the message, and reads the DAL into the Local Memory DAL area.
6.	DIOP	ACOM recognizes function code 1 as a new disk request. If the DSU is not busy, or if the queue is sufficiently short, ACOM calls subroutine DBUD to build executable DALs.
7.	DIOP	DBUD builds a DAL, allocates a disk buffer area in Buffer Memory, puts the DAL on the executable DAL queue, recognizes the request as a write, and sends a request to the BIOP to get data from the target memory. This request has a function code of 3 in the DAL; a copy of the DAL is written to Buffer Memory so the BIOP can move the data from the target memory to the Buffer Memory address specified in the DAL.
8.	BIOP	The interrupt handler gets 1 parcel from the accumulator channel and puts it on the ACOM data queue. The ACOM Activity Descriptor (AD) is put on the IOP central processor queue.
9.	BIOP	ACOM gets the entry from its queue. It recognizes function code 3, reads data from the target memory over the 100-Mbyte channel, and writes it to Buffer Memory.
10.	BIOP	ACOM sets the function code in the DAL to 6, writes it to the Buffer Memory assigned for this DAL, and sends an accumulator channel message to the DIOP.
11.	DIOP	The DIOP gets the interrupt on the accumulator channel and activates ACOM.
12.	DIOP	ACOM recognizes function code 6. The data is flagged, identifying its location as Buffer Memory.

Step	IOP	Description
13.	DIOP	If write-behind (meaning early status is requested) is specified and if this is the last sector for this request, status is sent to the MIOP and hence to the mainframe indicating that the data is in Buffer Memory.
14.	DIOP	If this sector is the first or second on the disk request queue for this channel, the data is read into Local Memory and flagged as Local Memory resident.
15.	DIOP	If this request is at the top of the queue, the write to disk begins. The Disk Control Block (DCB) is flagged as performing a write.
16.	DIOP	An interrupt indicates that the write is complete. The interrupt handler moves the top request on the ready queue to the done queue for the DCB. If the next I/O request is ready, the function is started. The DISK demon overlay is activated.
17.	DIOP	The DISK demon overlay takes the request off the done queue and releases the Buffer Memory space assigned. It checks to see whether this is the last write for the request; if it is, the DISK demon writes the master DAL to Buffer Memory with a code of 5, indicating that the status is to be returned to the mainframe. An accumulator channel message is sent to the MIOP.
18.	MIOP	The MIOP gets the interrupt, puts the accumulator channel data on the ACOM queue, and activates ACOM.
19.	MIOP	ACOM recognizes the function code of 5 and sends the DAL in six 64-bit words of data across the 6-Mbyte channel to the mainframe.
20.	MIOP	ACOM releases Buffer Memory space for the DAL and any Local Memory space.

3.2.4 STEPFLOW FOR DCU-4 DISK READ REQUEST FROM MAINFRAME

The following sequence of operations accomplishes a request to read a disk attached to the DIOP. The Target Memory Processor is BIOP. The IOP in which each step occurs is identified.

Step	IOP	Description
1.	MIOP	An interrupt on the 6-Mbyte channel is recognized as a disk request by interrupt answering. The CDEM overlay is activated.
2.	MIOP	CDEM forms a DAL using 30_8 parcels of information from the mainframe and 10_8 parcels of control information. The DAL function code is set to 1 (in DA@IFC) and put on the channel queue that connects IOPs, targeted for the DIOP. Buffer Memory is allocated, and the address is saved in the DAL. DA@MES contains the encoded address in Buffer Memory of the message. (See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for an example of a DAL format.)
3.	MIOP	A message crosses the accumulator channel to the DIOP.
4.	DIOP	The interrupt handler gets the accumulator channel data and puts it on the ACOM activities data queue. ACOM is activated by being placed on the IOP central processor queue.
5.	DIOP	ACOM gets the entry from the queue, calculates the Buffer Memory address of the message, and reads the DAL into the Local Memory DAL area.
6.	DIOP	ACOM recognizes function code 1 as a new disk request. If the disk unit is not busy, or if the queue is sufficiently short, ACOM calls subroutine DBUD to build executable DALs.
7.	DIOP	DBUD builds a DAL, allocates disk space, and queues the request on the executable DAL queue. The request is recognized as a read. If the entry is the first on the queue, the I/O is initiated on the proper disk channel.
8.	DIOP	An interrupt indicates that I/O has successfully completed. The interrupt handler moves the DAL to the done queue and starts the next DAL (if another exists) or begins read-ahead (if last). The DISK demon overlay is activated.
9.	DIOP	The DISK demon overlay takes the DAL off the done queue and moves data from Local Memory to Buffer Memory. The executable DAL, with a function code of 4, is written to Buffer Memory and its address is passed to BIOP across the accumulator channel.
10.	BIOP	The interrupt handler gets the accumulator, puts it on the ACOM queue, and activates ACOM.

Step	IOP	Description
11.	BIOP	ACOM gets the entry from the queue. It recognizes function code 4, reads data from Buffer Memory, and writes it to the target memory through the 100-Mbyte channel.
12.	BIOP	ACOM sets the function code in the DAL to 7, writes it to Buffer Memory assigned for the DAL, and sends the accumulator channel message to the DIOP.
13.	DIOP	The interrupt handler gets an interrupt on the accumulator channel, queues the message, and activates the ACOM activity.
14.	DIOP	ACOM recognizes function code 7. It releases Buffer Memory and DAL space. If this is the last DAL for this request, ACOM puts a function code of 5 into the master DAL (indicating status is to be sent to the mainframe), writes the DAL to Buffer Memory, and sends an accumulator channel message to the MIOP.
15.	MIOP	The MIOP gets the interrupt, puts the accumulator channel data on the ACOM queue, and activates ACOM.
16.	MIOP	ACOM recognizes the function code of 5 and sends the command from the DAL (containing status) to the mainframe across the 6-Mbyte channel.
17.	MIOP	ACOM releases Buffer Memory space for the DAL and any Local Memory used.

3.2.5 LOCAL HANDLING OF DISK QUEUES

Each IOP maintains its own queues in Local Memory (by disk unit) for disks attached to it.

Queues are composed of linked lists of disk packets that take the same form as the 30_8-parcel packets, or DALs, that come from the mainframe. These packets reside in Buffer Memory and are allocated and deallocated by the MIOP. When data for a disk I/O comes into Local Memory (from Buffer Memory or disk), a copy of the packet has already been read into Local Memory. When I/O is complete, the packet is removed from the queue and its address sent to the next IOP, which performs processing for it.

Disk data must always move through the Local Memory of the processor attached to the relevant disk.

Each IOP facilitates streaming on devices by reading ahead as requested by disk packets and by maintaining the data read-ahead in Buffer Memory until subsequent requests are received for this data. The data in these read-ahead buffers is retained in Buffer Memory. If no request is received to transfer the data to the mainframe before the next write request, the Buffer Memory is deallocated for use by other requests. Each read request that is received causes examination of the read-ahead queue to find data that has been preread.

3.2.6 DCU-4 DISK READ-AHEAD

Each disk control block (DCB) has a Read-ahead Control Table assigned to it at initialization time. These tables are used on both disk reads and writes to anticipate subsequent I/O requests.

Each entry in a Read-ahead Control Table represents one sector of data and contains information used to identify and locate that sector. The number of entries in each Read-ahead Control Table is determined by the $APTEXT constants RA$NUM (single-device units) and RA$NUMS (striped-device units). Entries in the Read-ahead Control Table are described by the RA@ definitions (see the IOS Table Descriptions Internal Reference Manual, publication SM-0007).

The Read-ahead Control Table is treated as a circular list and has associated with it the following pointers located in the DCB:

Pointer	Description
DB@INF	Pointer to the first entry in the table; remains constant from initialization.
DB@UNF	Pointer to the last entry in the table; remains constant from initialization.
DB@IN	Pointer to the next entry to be read into (reading if from disk to Buffer Memory; writing if from the target memory to Buffer Memory).
DB@OUT	Pointer to the next entry to be used to satisfy a request
DB@RAK	Count of number of entries in the Read-ahead Control Table; remains constant from initialization.
DB@RAF	Count of number of entries containing read-ahead data

3.2.6.1 Disk read

Upon completion of a disk read request, the read interrupt routine checks for additional I/O requests for the current channel. If there are none, read-ahead control is initiated. Read-ahead control anticipates that the next I/O request for the current channel will be a read for sectors contiguous to the one just read.

Two cases in which read-ahead is not initiated and the channel is idled until the next I/O request is received areas follows:

- When a seek is required to position to the next sequential sector
- When no local disk buffers are available

The read-ahead sequence is as follows:

1. Read complete. No requests pending (DB@MDL=0). Initialize Read-ahead Control Table:
   DB@RAF=0
   DB@IN=DB@OUT=DB@INF
2. Compute disk address of next sector to be read and store in the entry pointed to by DB@IN (RA@CYL, RA@HED, and RA@SEC).
3. Allocate a local disk buffer and store the address in the read-ahead entry (RA@LOC).
4. Set disk activity in DCB (DB@FLG) to read-ahead (function code is 4) and initiate read.

3.2.6.1.1 Interrupt occurs due to read-ahead - Read-aheads are terminated if any of the following conditions are found:

- Read-ahead abort is set in the DCB (bit 2^{15} in DB@FLG).
- An error is detected in the read-ahead just completed.
- No more entries are available in the Read-ahead Control Table (DB@RAF=DB@RAK).
- No local disk buffers are available.
- A seek is required to position to the next sequential sector.

The following sequence applies to an interrupt due to a read-ahead:

1. Set data location in entry pointed to by DB@IN to Local Memory (RA@DAT=1). Increment count of read-ahead entries in use (DB@RAF+1). Advance DB@IN to next entry.

2. Compute next sequential disk address and store in the entry pointed to by DB@IN (RA@CYL, RA@HED, and RA@SEC). Allocate a local disk buffer and store the address in the read-ahead entry (RA@LOC). Initiate read.

3. Activate Disk Demon to move read-ahead sector just completed to Buffer Memory.

3.2.6.1.2 Disk Demon read-ahead process - The Disk Demon read-ahead process follows:

1. Allocate Buffer Memory buffer and save address in the entry pointed to by DB@IN (RA@BM0/BM1).

2. Transfer data from Local Memory buffer (RA@LOC) to Buffer Memory. Set data location to Buffer Memory (RA@DAT=2). Release Local Memory buffer.

3.2.6.1.3 I/O request received during read-ahead - If the I/O request received specifies a write, the Read-ahead Abort flag is set (2^{15} in DB@FLG), causing read-aheads to terminate after the current read completes.

If the I/O request received specifies a read, a check is made to see whether the first sector of the request matches either data previously read-ahead or the sector currently being read.

ACOM checks to see if the first sector of the request matches any of the sectors previously read-ahead. If there is a match, ACOM makes sure that the pointer DB@OUT gets set to the matching entry.

ACOM then calls the DBUD routine to prepare E-DALs for the request. As each E-DAL is built, a check is made to see whether the sector for which the E-DAL is being built matches the sector in the read-ahead entry pointed to by DB@OUT. If there is a match, the memory location of the data is moved from the read-ahead entry to the E-DAL. The E-DAL is then placed on the done queue (DB@DNQ) for processing by the Disk Demon. The DB@OUT pointer is then advanced to the next read-ahead entry and the read-ahead count (DB@RAF) is decremented.

This process continues until all of the read-ahead data has been used.

If the read-ahead count (DB@DRAF) is 0 and read-ahead is in progress (DB@FLG=4), a check is made to see if the sector for which the E-DAL is being built is the same as that being read. If they are the same, the information from the read-ahead entry pointed to by DB@IN is moved to the E-DAL, the Disk Activity flag is set to read (DB@FLG=1), and the E-DAL is put on the executable queue (DB@EDL). This process is referred to as a read-ahead steal, because the active read-ahead is redirected into the normal read flow.

3.2.6.2 Disk write

Read-ahead during a disk write (sometimes referred to as write-behind) involves moving sectors of data from the target memory to Buffer Memory, where it is held until the disk is positioned to where the data is to be written. A response is sent after the last sector of data (for a given request) has been moved out of the target memory. The read-ahead allows overlap of current and previous request processing; that is, overlap of the preparing of data to be written with the writing of data to disk.

Read-ahead during a disk write always attempts to keep the Read-ahead Control Table full. As each write request is satisfied and a new one received, data is moved to Buffer Memory, as long as entries are available in the Read-ahead Control Table. As each sector is moved from Buffer Memory to Local Memory to disk, the next sequential sector of the most recent request is moved from the target memory to Buffer Memory to take its place.

3.2.6.2.1 Disk Demon: Read-ahead input - The sequence that applies to Disk Demon read-ahead input follows:

1. Check to see whether there is any more room in the Read-ahead Control Table (DB@RAF=DB@RAK). If no room exists, the sequence ends here.

2. Check each M-DAL queued (DB@MDL) beginning with the first for a sector waiting to be moved. (DA@WBH # DA@TOT). If none exist, the sequence ends here.

3. Transfer sector from the target memory to Buffer Memory. Save Buffer Memory address of the entry in the Read-ahead Control Table that is pointed to by DB@IN (RA@BM0/BM1). Save the disk address of the sector in the entry (RA@CYL, RA@HED, and RA@SEC).

4. Advance DB@IN to the next entry. Increment read-ahead count (DB@RAF). Increment next read-ahead/write-behind sector in the M-DAL (DA@WBH).

5. If the last sector for the request has been moved (DA@WBH=DA@TOT), send a response to the mainframe.

3.2.6.2.2 Disk Demon: Read-ahead output - The sequence that applies to Disk Demon read-ahead output follows:

1. Call the Kernel DBUD routine to build the next E-DAL.

2. DBUD checks to see if the read-ahead entry pointed to by DB@OUT matches the sector for which the E-DAL is being built.

3. The Buffer Memory address is moved from the entry (RA@BM0 and RA@BM1) to the E-DAL (DA@BM0 and DA@BM1). The data location is set to Buffer Memory in the E-DAL (DA@DAT=2). DB@OUT is advanced to the next entry and the read-ahead count is decremented (DB@RAF).

4. Disk Demon allocates a local buffer, detects that data is in Buffer Memory rather than target memory (DA@DAT=2), and transfers data from Buffer Memory to Local Memory in preparation for a write to disk.

3.2.7 ON-LINE DISK DIAGNOSTIC REQUESTS

The DCU-4 disk driver supports on-line disk diagnostic requests for the data, format, and correction code options of the read/write commands. In addition to specifying a read/write option, the diagnostic request may enable or disable error recovery, error reporting, read-ahead, and write-behind.

Diagnostic request processing in the IOS proceeds according to the following rules:

- Only full sector I/O is supported
- Diagnostic requests may only be made for physical devices. Thus, a diagnostic request for a device that is a member of a striped group is valid, while a request for the striped group itself is invalid.

In the event an error occurs during diagnostic request processing, the IOS returns an error record to the diagnostic job and/or the system log if indicated in the original request packet. See subsection 3.3.4, Error Status Returned to Mainframe, for more information on error reporting.

3.3 DCU-4 DISK ERROR RECOVERY

The Kernel's disk error recovery routines process and recover from errors on disk. The routines are resident in the Kernel overlay ERRECK, which is activated when a disk error is recognized on one of the IOP disk channels. The DISK overlay creates ERRECK when it recognizes that error recovery must be attempted.

After activation, the ERRECK overlay attempts recovery in a predefined order, according to entries in a Kernel table and depending on the type of error. Table 3-4 summarizes the process.

ERRECK recognizes the following four types of disk errors:

- Data errors, indicating the data was not transferred correctly
- Lost data errors, indicating memory was unable to keep up with the disk data transfers
- Seek errors, resulting from the incorrect physical movement of the read/write head
- Disk interlock, which occurs when the disk is not physically ready to transfer data

The Kernel maintains statistics for each disk unit on the number and types of errors for each disk channel.

3.3.1 DISK ERRORS REQUIRING RECOVERY

Disk storage units signal an error condition by setting the done bit and leaving the busy bit set on the channel. The done bit causes an interrupt that activates the ERRECK routine. The done and busy condition is sensed by the disk interrupt routine, which reads the status into the A register from the channel to determine the type of error that has occurred. Table 3-1 shows the relationship between the contents of the A register and the error condition. A bit signals an error if it is set to 1. These errors are defined in the following subsections.

Table 3-1. Error Conditions

Bit	Error
0	Angular position counter failure
1	Lost function
2	Lost data
3	Read error channel 3
4	Read error channel 2
5	Read error channel 1
6	Read error channel 0
7	Address error
8	Seek error
9	Write error channel 3
10	Write error channel 2
11	Write error channel 1
12	Write error channel 0
13	Multiple head select
14	Read/write conflict
15	Read/write off cylinder

3.3.1.1 Data error

Data errors are detected on read and write functions when the hardware senses that the correct data has not been transferred as requested. The Kernel disk interrupt answering routine senses that both Done and Busy flags are set and reads the disk status. If any of the bits between 3 and 6 are set in the status parcel, an error occurred in transferring data from the disk. If any of the bits between 9 and 12 are set, an error occurred when trying to transfer data to the disk.

3.3.1.1.1 Recovery for data errors on read operations - When a data error is encountered, the Kernel tries to recover the data with a series of operations. The recovery sequence occurs in the following order:

1. Error recovery repeats the read operation a fixed number of times to determine if the error is transient.

2. If the function repetition fails, recovery is attempted through cylinder margin selection, read early/late selection, or combinations of the two. The READSEQ table in ERRECK controls the sequence of events and contains margin and read early/late parameters.

3. Disk error correction is attempted for data errors if cylinder margin and read early/late selection retries are unsuccessful. Error recovery reads the data and the associated error correction code without cylinder offset or read early/late selection. The overlay FIRECODE is called to generate correction vectors and correct the data, if possible. The error correction algorithm corrects data in a single burst of 11 bits or less for each of the four read heads.

   The disk error correction feature can be disabled if desired (see the I@IOSECC parameter description in the COS Operational Procedures Reference Manual, SM-0043 or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018).

4. If none of the preceding procedures is successful, error recovery sends the sector of data containing the error to the mainframe along with a status indicating the unsuccessful data request. The remainder of the current disk request is thrown away, and the Kernel continues processing any subsequent requests.

3.3.1.1.2 Recovery for data errors on write operations - If the disk hardware detects an error while attempting to write data to disk, the error recovery routine repeats the function a set number of times to determine if the error is transient. If the requests are not successful, the IOS returns a status to the mainframe indicating unsuccessful completion of the operation.

3.3.1.2 Lost data errors

When the status parcel has a 1 set in bit 2, the hardware has detected that Local Memory was unable to keep up with the disk transfer on a read operation. In this case, the data transfer was not completed, and error recovery attempts to complete the function by repeating it a set number of times.

ERRECK then clears fault flags, does a seek to cylinder 0, and attempts to repeat the disk function. This operation is repeated a set number of times. If the data is not successfully transferred by these repeated operations, the IOS returns a status to the mainframe indicating unsuccessful completion of the operation.

3.3.1.3 Seek errors

Seek errors are detected by the hardware and are indicated when bit 8 is set in the status parcel. The recovery procedure is to return to cylinder 0, then attempt to do the seek again. This sequence is repeated a set number of times. If the seek cannot be completed successfully, an error status is returned to the mainframe.

3.3.1.4 ID errors

Following a normal disk seek operation, the hardware returns the cylinder number from the disk ID field in the Status Response register. If this cylinder number does not agree with the cylinder that software is trying to select, error recovery is invoked. The error recovery procedure is to return to cylinder 0, then attempt to do the seek again. This sequence is repeated a set number of times. Before the final retry, the head group is switched in an effort to determine if the correct cylinder is being selected. If all retries fail, an error status is returned to the mainframe.

3.3.1.5 Interlock status

When error recovery finds no bits set in the status parcel after detecting an error condition, it knows that the disk referenced is not in a condition to perform the I/O. To determine the cause of the condition, the error recovery overlay reads the interlock status into the status response register and then into the A register with an IOB:11 instruction. Error recovery checks to see whether the IOS has reserved bit set. If reserved, the status word is checked to see if a real interlock condition is set. If not set, the recovery routine considers the interlock false and tries to recover as though it were a miscellaneous type. Otherwise, error recovery displays a message indicating the type of error so the operator can correct physical interlocks. An interlock status (irrecoverable error) is returned to the mainframe.

Conditions considered interlocks, along with their bit positions in the status response register, are indicated in table 3-2. In all cases, a 1 in the bit position indicates that the corresponding condition is true.

Table 3-2. Interlock Error Conditions

Bit	Error
8	Positive voltage supply for the DSU is below normal
9	Negative voltage supply for the DSU is below normal
11	DSU start switch is off
12	DSU brush cycle is in process
13	Disk heads are not loaded on the disk surface
14	Disk surface is not up to speed
15	Disk drive cabinet is over the normal temperature range

3.3.1.6 Miscellaneous disk errors

Certain disk errors do not fit neatly into any of the previous classifications. When these errors occur, they are treated as transient conditions that may disappear on retry, and the last function executed on the channel is reexecuted up to a set maximum number of times. If the error continues to occur, the condition is processed as though it were an interlock condition, causing a message to be sent to the operator and a status response to the mainframe.

Miscellaneous errors, along with their bit positions in the status response register, are given in table 3-3. A 1 in the bit position indicates that the condition is true.

Table 3-3. Miscellaneous Error Conditions

Bit	Error
0	Angular position counter failure
7	Address error
13	Multiple head select
14	Read and write conflict
15	Read/write off cylinder

3.3.2 I/O TIME-OUT

When a read, write, or seek function is sent to a disk channel, the timer entry for that channel (DB@TMO) is passed along with a time-out value (SEEKLIM/SEEK, DISKTLIM/READ, or WRITE) to the QTIME routine for placement on the system event timer queue. If the interrupt occurs before the timer expires, a call is made to DQTIME to remove the entry from the timer queue.

If the interrupt does not occur in time, control is given to the IDKTOUT routine. IDKTOUT is entered into TMR@RT of each timer entry (DB@TMO) for all disk channels at system initialization. The IDKTOUT routine either activates the disk demon (DISK) to initiate error recovery or activates ERRECK if error recovery is already in progress.

3.3.3 ERROR RECOVERY SUMMARY

Table 3-4 summarizes the handling of the various disk error conditions. The order of recovery is defined by the lowercase letters; the letter *a* designates the first operation attempted; *b*, the second, and so on. The recovery actions are abbreviated as follows:

R	Repeat last function	I-M	IOP message to CRT
M	Margin select	ST	Send status to mainframe
E/L	Strobe early and late	C0	Return to cylinder 0 and retry
C	Combination of M and E/L	FC	Firecode processing (error correction code)
RS	Read interlock status		

Table 3-4. Disk Error Recovery Summary

Error Condition	Bit	R	M	E/L	C	RS	I-M	ST	C0	FC
Angular position failure	0	a					c	d	b	
Disk not ready	1	a					c	d	b	
Lost data	2	a					c	d	b	
Data error, channel 3	3	a	b	c	d		f	g		e
Data error, channel 2	4	a	b	c	d		f	g		e
Data error, channel 1	5	a	b	c	d		f	g		e
Data error, channel 0	6	a	b	c	d		f	g		e
Address error	7	a					b	c		
Seek error	8	b					c	d	a	
Write fault, channel 3	9	a					c	d	b	
Write fault, channel 2	10	a					c	d	b	
Write fault, channel 1	11	a					c	d	b	
Write fault, channel 0	12	a					c	d	b	
Multiple head select	13	a					c	d	b	
Read/write conflict	14	a					c	d	b	
Read/write off cylinder	15	a					c	d	b	
Time-out		a					b	c		
ID error		b					c	d	a	
Interlock		b				a	c	d		

3.3.4 ERROR STATUS RETURNED TO MAINFRAME

When a DD-19 or DD-29 disk error occurs, the IOS returns the final error status to the mainframe. The status is returned in field DA@RC of the disk command packet that made the original request. Valid error statuses that may be returned are as follows:

Error Status	Description
DAR$OK	No error encountered
DAR$REC	Recovered error
DAR$COR	Corrected data error
DAR$UNC	Uncorrected data error
DAR$UNR	Unrecovered error

Limited information about disk errors is passed back to COS in parcels 30_8 through 33_8 of the DAL (parcels 20_8 through 23_8 of data transferred to the mainframe). Table 3-5 defines this information.

Table 3-5. Disk Error Information in DAL

Parcel	Bits	Description
30	0-10	Cylinder on which error was detected
30	11-15	Head group in which error was detected
31	0-6	Sector of error
31	7-15	Offset from the beginning of the sector
32-33	0-15	Length of actual transfer

Additional information is sent to the mainframe in a disk error packet for logging. This packet is built in the REPORT overlay. (See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for the format of the Disk Error Packet, DE@.)

3.3.5 DCU-4 DISK ERROR MESSAGE

If an irrecoverable error occurs in an IOP, the error reporting overlay (REPORT) informs the operator of the error location, the hardware status returned, and time of the error through the following message:

CHANNEL *chan* - DISK ERROR CYL *cyl* HD *hd* ST *status hh:mm:ss*

chan Channel number (20 through 37_8)

cyl Cylinder number

hd Head number

status If hardware detected the error, *status* is the hardware status code returned from disk. If software detected the error, *status* is one of the following:

INT-LK	Channel interlocked, not ready
BAD-SK	Bad cylinder number after seek (ID error)
TM-OUT	Channel timed out

hh:mm:ss IOP time of day

3.4 DCU-5 DISK CONTROLLING SOFTWARE

This subsection describes the architecture and request process for the DCU-5 (RD-10, DD-39, DD-40, and DD-49) controlling software.

The DD-40 DSUs are part of the DS-40 Disk Subsystem. The DD-40s are connected to the DCU-5 through DC-40 Disk Controllers. The RD-10 DSUs are part of the RD-10 Transportable Disk Subsystem and are connected to the DCU-5 through a DCU-S1 serial Disk Controller.

It is recommended that RD-10s not share the same DCU-5 controller as other disk types, as this may result in overrun/underrun errors on the RD-10s during periods of heavy activity. These errors are recoverable, but will result in a loss of performance and a potentially large number of errors being logged.

3.4.1 DCU-5 SOFTWARE COMPONENTS

The DCU-5 disk controlling software consists of the following four demon overlays and the interrupt answering overlay:

Overlay	Function
DD49	Loads into Local Memory at IOS deadstart time and resides as the disk interrupt handler in the IOPs that have DCU-5 disk devices configured.
D4DEM	Starts I/O to the disk and performs tasks to keep the I/O going
ICOM	Handles all communication between IOPs for the DCU-5 disk driving software
TRANSFR	Moves disk data between Buffer Memory and the target memory through the TRANSFER Kernel service request

3.4.2 DCU-5 DISK DRIVER TABLES AND PACKETS

The following are software components that control the disk processing. These structures are defined in the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

3.4.2.1 Disk Request Packet (DAL) - DL@

The DAL is a fixed-length packet containing request information from the mainframe plus control information used by the IOS. For more information about DALs, see subsection 2.14, MIOP-mainframe Communication Channel.

3.4.2.2 Disk Control Block (DCB) - DK@

The DCB is the main control table for disk operations. One DCB is defined for each disk channel. This table is created at system initialization and resides in Local Memory.

3.4.2.3 Local Buffer entry - LB@

The Local Buffer entry is used for Local Memory buffer control. There is one entry for each dedicated Local Memory buffer being used by the disk channel. The entries serve as requests to the D4DEM overlay to move a sector of data to or from Local Memory. The Local Buffer entries immediately follow the DCB.

3.4.2.4 Buffer Memory Control Block (MCB) - CB@

The MCB is a table resident in Buffer Memory. The MCB controls the flow of data between the target memory and the disk on IOPs without a 100-Mbyte channel to that target memory. There is one MCB for each disk channel defined. This table is created at system initialization. A pointer to the MCB is contained in its associated DCB in Local Memory.

3.4.2.5 Data Transfer Request (DTR) - TR@

The DTR is a fixed-length packet used to make requests to the TRANSFR overlay. It contains all the necessary information to move data between Buffer Memory and target memory.

3.4.2.6 Abort Transfer Request (ATR) - AR@

The ATR is a fixed-length packet used to terminate the movement of data between Buffer Memory and the target memory before the completion of a DTR. This packet is sent to the TRANSFR overlay in the event of an irrecoverable disk error.

3.4.2.7 Device Parameter Table (DPT) - DP@

The DPT contains information common to all disk devices of the same type. During system initialization, one DPT is established in Local Memory for each configured device type. Each DCB contains a pointer to the DPT associated with its device type.

3.4.2.8 MEMIO Queue Table - MEM@

The MEMIO queues serve as request queues for all disk channels needing data moved to or from Local Memory. Each request is a Local Buffer entry. Each memory channel has its own distinct queue, allowing for I/O overlap on the different channels. This table is serviced by the D4DEM overlay.

3.4.3 RESOURCE MANAGEMENT

The DCU-5 disk driving software makes extensive use of two IOS resources: Local Memory and Buffer Memory.

3.4.3.1 Local Memory management

The sector chaining feature requires at least two dedicated Local Memory disk buffers per active disk channel. The number of buffers is controlled by the equates LB$DD10 for RD-10 disks, LB$DD39 for DD-39 disks, LB$DD40 for DD-40, and LB$DD49 for DD-49 disks.

Local buffers are allocated when a request is received to activate a disk channel and are retained for as long as that channel is busy. Each buffer address is assigned to a Local Buffer entry. These buffers are used in a circular fashion to store data on its way to or from disk. Buffer allocation and release are performed by D4DEM.

3.4.3.2 Buffer Memory management

Each DCU-5 disk channel has a number of contiguous Buffer Memory disk buffers assigned to it at IOS initialization time. These buffers are used in a circular fashion for read-ahead and write-behind data. For disks on IOPs with a 100-Mbyte channel, the starting buffer address and control information is stored in the channel's DCB. For disks on IOPs without 100-Mbyte channels, the address and information is stored in the channel's MCB. The number of buffers allocated is based on the read-ahead and write-behind count constants. More detailed information on read-ahead and write-behind is included later in this section.

3.4.4 DCU-5 DISK READ REQUEST STEPFLOW

The following sequence of operations handles a read request to a DCU-5 type DSU attached to the DIOP:

1. CDEM receives a request packet (DAL) from the mainframe, recognizes it as a DCU-5 disk request, and routes it to ICOM in the target IOP. See discussions of the MIOP-mainframe Communication Channel and Communication Between IOPs in section 2 for more information about packet handling.

2. ICOM in the target IOP validates the DAL parameters, puts the DAL on the DCB DAL queue, and activates D4DEM.

3. D4DEM allocates Local Memory buffers and starts I/O to the disk. If this IOP does not have a 100-Mbyte channel to the specified target memory, a DTR is sent to TRANSFR in the Target Memory Processor.

4. An interrupt signals the completion of a sector transfer from disk. The DD49 interrupt handler continues the next sector I/O to disk and initiates the I/O to empty the Local Memory buffer just filled with disk data. This sector of data is moved from Local Memory to the target memory across the 100-Mbyte channel while the disk is filling the next Local Memory buffer with data. If this IOP does not have a 100-Mbyte channel, the data is moved to Buffer Memory where TRANSFR can then move it to the target memory. DD49 then activates D4DEM.

5. D4DEM waits for the sector transfer to the target memory to complete, and prepares the Local Memory buffer for the next read from disk.

6. Steps 4 and 5 are repeated until the entire request is complete. When all the data has moved to the target memory, D4DEM is activated either by DD49, if the IOP has a 100-Mbyte channel, or by ICOM when ICOM receives a message from TRANSFR indicating the DTR has been completed.

7. D4DEM sends the final status to ICOM in MIOP and releases the Local Memory buffers.

8. ICOM in MIOP sends the DAL containing status back to the mainframe to signal completion of the request.

3.4.5 DCU-5 DISK WRITE REQUEST STEPFLOW

The following sequence of operations handles a write request to a DCU-5 type DSU attached to the DIOP.

1. CDEM receives a request packet (DAL) from the mainframe, recognizes it as a DCU-5 disk request, and routes it to ICOM in the target IOP. See subsection 2.13, Communication Among IOPs, and subsection 2.14, MIOP-mainframe Communication Channel, for more information about packet handling.

2. ICOM in the target IOP validates the DAL parameters, puts the DAL on the DCB DAL queue, and activates D4DEM.

3. If this IOP has a 100-Mbyte channel to the specified target memory, D4DEM allocates Local Memory buffers, and fills the first buffers with data from the target memory across the 100-Mbyte channel. If this IOP does not have a 100-Mbyte channel, D4DEM sends a DTR to TRANSFR in the target memory processor. The data is obtained from Buffer Memory after TRANSFR has made it available.

4. D4DEM initiates I/O to the disk.

5. An interrupt signals the completion of a sector transfer to disk. The DD49 interrupt handler continues the next sector I/O to disk and initiates the I/O to fill the next available Local Memory buffer with data. This sector of data is moved from the target memory to Local Memory across the 100-Mbyte channel while the data from the current Local Memory buffer is emptied to disk. If this IOP does not have a 100-Mbyte channel, the data is moved from Buffer Memory after TRANSFR has obtained it from the target memory. DD49 then activates D4DEM.

6. D4DEM waits for the sector transfer from the target memory (or Buffer Memory) to complete, and prepares the Local Memory buffer for the next write to disk.

7. Steps 5 and 6 are repeated until the entire request is complete. At that time, D4DEM is activated by the DD49 interrupt handler.

8. D4DEM sends the final status to ICOM in MIOP and releases Local Memory buffers.

9. ICOM in MIOP sends the DAL containing status back to the mainframe to signal completion of the request.

3.4.6 DCU-5 READ-AHEAD AND WRITE-BEHIND

The DCU-5 driving software uses Buffer Memory as a disk cache in anticipation of contiguous disk requests from the mainframe. This subsection describes the cache mechanism.

3.4.6.1 DCU-5 read-ahead

Upon completion of a disk read request, the DD-49 interrupt handler automatically initiates read-ahead I/O. Read-ahead control anticipates that the next I/O request for that channel will be a read for sectors contiguous to the previous request.

Read-aheads are terminated if any one of the following conditions occur:

- The read-ahead count, defined in $APTEXT, is satisfied; the read-ahead equate takes the form RA$*type*, where *type* indicates the device type.

- A seek is required to position to the next sequential sector

- Sector chaining is broken

- An error is detected
- The Read-ahead Abort flag, DK@ABT, is set in the DCB; this flag is set when an I/O request is received that cannot be satisfied by the current read-ahead sectors.

When an interrupt occurs, signaling the completion of a read-ahead, the DD49 interrupt handler activates D4DEM to store the sector of data in the Buffer Memory read-ahead area.

If the next I/O request cannot be satisfied by the current read-ahead data, read-aheads are terminated, as required. D4DEM then updates the pointers to the Buffer Memory read-ahead area, indicating the read-ahead data has been thrown away.

If the next I/O request can be satisfied by the current read-ahead data, D4DEM moves the data stored in Buffer Memory to target memory. Any remaining sectors for the request are read from disk through the normal path.

3.4.6.2 DCU-5 write-behind

Write-behind control during write request processing is the equivalent of read-ahead control during read request processing. It involves storing sectors of data in Buffer Memory until the disk can be positioned where the data is to be written. The mainframe is notified when all data for the current write request has been moved out of the target memory. This allows for overlapping the preparation of the next I/O request with the writing of data to disk for the current request.

Write-behind processing occurs only under the following conditions:

- The write-behind count constant, defined in $APTEXT, has not been satisfied; the write-behind equate takes the form WB$*type*, where *type* indicates the device type.
- All local buffers are full
- There is data remaining in the target memory for the most recent write request
- There are no pending sectors on queue to be moved to or from Local Memory buffers

D4DEM attempts to satisfy the write-behind count by transferring sectors for the most recent request from the target memory to Buffer Memory in reverse order; for example, the last sector for a request is moved into Buffer Memory first, the next-to-last sector is moved second, and so on.

When the DD-49 interrupt handler detects that the next sector of data resides in Buffer Memory rather than the target memory, it activates D4DEM to get the data from the appropriate location. D4DEM updates the pointers to the Buffer Memory write-behind area after moving the last sector for a request into Local Memory, indicating the Buffer Memory space is now free for subsequent write-behind requests.

3.4.7 SPIRAL FORMATTING

The DCU-5 disk driver incorporates spiral formatting to reduce the time spent waiting for cylinder-to-cylinder seeks to complete. Spiral formatting is done by the software and does not affect physical formatting of the drive. Only RD-10, DD-39 and DD-49 disk drives are spirally formatted. DD-40 disk drives are not spirally formatted as this would defeat the read-ahead and write-behind logic in the DC-40 controller.

On a disk with spiral formatting, each cylinder starts a partial revolution later than the previous cylinder. Thus, when moving from one cylinder to the next, sector 0 of the new cylinder is available for reading within a partial revolution, rather than waiting for a full revolution to complete.

To implement spiral formatting, the driver maps each logical data sector onto a physical sector according to a conversion table residing in the DPT for that device. This table contains a list of offset values used to calculate the physical sector number.

To calculate the physical sector number, add the value of the low-order bits of the requested cylinder number to the base address of the conversion table. The resulting location contains the correct offset value. Add the offset to the logical sector number to obtain the physical sector number.

Only the low-order bit of the cylinder number is used in the conversion algorithm for DD-10s and DD-39s; thus the disk is logically divided into two halves. The low-order 2 bits of the cylinder number are used in the conversion algorithm for DD-49s; thus, the disk is logically divided into four quadrants.

3.4.8 ON-LINE DISK DIAGNOSTIC REQUESTS

The DCU-5 disk driver supports on-line disk diagnostic requests for all options of the read/write commands. In addition to specifying a read/write option, the diagnostic request may enable/disable error recovery, error reporting, read-ahead, and write-behind.

Diagnostic request processing in the IOS proceeds according to the following rules:

- Only full sector I/O is supported
- Diagnostic requests may be made only for physical devices. This means that a diagnostic request for a device that is a member of a striped group is valid, while a request for the striped group itself is invalid.
- Software spiral formatting is not part of diagnostic request processing.

If an error occurs during diagnostic request processing, the IOS returns an error record to the diagnostic job and/or the system log, if so indicated in the original request packet. See subsection 3.5.4, Error Reporting, for more information.

3.5 DCU-5 DISK ERROR RECOVERY

Error recovery for the DCU-5 type disks is enacted when the DD-49 disk interrupt handler detects one of the following error types: software detected, status register 0, or Drive General Status. When the DD-49 disk interrupt handler detects an error on a disk channel, an error status is stored in the DCB for the channel and the D4DEM overlay is activated. D4DEM recognizes that an error has occurred and creates an error recovery activity for the channel.

Error recovery consists of five major areas that are assumed to be single processes; it is rarely required to go from one process to another during error recovery. These areas are as follows:

- Unit Select Process (D4SLR for DD-49 or D3SLR for RD-10, DD-39 and DD-40)
- Cylinder Select Process (D4SKR for DD-49 or D3SKR for RD-10, DD-39, and DD-40)
- Head Select-LMA Select-Read Process (D4IOR for DD-49, D40IOR for DD-40, D3IOR for DD-39, or D10IOR for RD-10)
- Head Select-LMA Select-Write Process (D4IOR for DD-49, D40IOR for DD-40, D3IOR for DD-39, or D10IOR for RD-10)
- Release Process (D4RLR for DD-49 or D3RLR for RD-10, DD-39, and DD-40)

The following subsection describes the overlays that make up the error recovery activity.

3.5.1 RECOVERY ACTIVITY

The error recovery activity consists of one controlling overlay, four overlays to perform each major recovery process, three overlays to perform subprocess disk functions, and two overlays to report errors.

The recovery activity is table-driven (for flexibility) and recursive so that the procedure can tolerate errors on the recovery functions. It is also time-delayed so that environmentally induced errors have a chance to dissipate without the DCU-5 software becoming dedicated to error recovery. An overall maximum retry count per I/O sector and error process is assigned (see the subsection 3.5.2, Error Recovery Process). Within each process a retry limit is assigned to the various subprocess errors, that is, errors which occur in the recovery functions themselves. See tables 3-6 and 3-7 for the retry limits on each recovery function.

For more information on disk hardware status, see the Disk Systems Hardware Reference Manual, publication HR-0077.

Table 3-6. DD-49 Error Retry Limits

Select	Seek	Read	Write	Release	Description
63	63	140	63	63	Maximum retries available
15	15	15	15	15	Time-out
15	15	15	15	15	Not ready
15	-	-	-	15	Busy response
15	15	15	15	-	Input parity error
15	15	15	15	-	Sequence option in progress
-	15	15	15	-	Invalid command or option, function or bus-out parity
-	15	15	15	-	Function lost
5	15	15	15	-	Catastrophic drive error
-	15	15	15	-	Seek fault
-	-	15	15	-	Overrun/underrun

Table 3-6. DD-49 Error Retry Limits (continued)

Select	Seek	Read	Write	Release	Description
-	-	45	3	-	ECC, ID not found, and synchronization time-out (retries per offset position)
-	-	1	1	-	Initial Local Memory Address (LMA) echo error
-	-	0	15	-	Final LMA echo error

Table 3-7. RD-10, DD-39, and DD-40 Error Retry Limits

Select	Seek	Read	Write	Release	Description
63	63	140	63	63	Maximum retries available
15	15	15	15	15	Time-out
15	15	15	15	15	Not ready
15	-	-	-	15	Busy response
15	15	15	15	-	Input parity error
-	15	15	15	-	Command error; sequencer function or bus-out parity.
15	15	15	15	-	Sequence option in progress
5	15	15	15	-	Catastrophic drive error
-	15	15	15	-	Seek fault
-	-	15	15	-	Overrun/underrun
-	-	45	3	-	ECC, ID not found, and synchronization time-out (retries per offset position)
-	-	1	1	-	Initial LMA echo error
-	-	0	15	-	Final LMA echo error

A description of each overlay in the recovery activity follows.

Overlay	Function
D4ERR/ D40ERR/ D3ERR	This overlay is the initial and controlling overlay for the error recovery activity. It determines the major error type, sets up recovery tables, and calls the appropriate overlay to process the error. At the completion of the recovery process, D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39) prepares the disk channel for subsequent requests and activates D4DEM to continue normal I/O processing.
D4ECC/ D40ECC/ D3ECC	This overlay performs error correction code for read data errors. It is called by D4IOR (DD-49), D40IOR (DD-40), D3IOR (DD-39), or D10IOR (RD-10) when a read data error is determined to be a good candidate for correction. The D40ECC overlay performs error correction for RD-10s.
D4IOR/ D40IOR/ D3IOR/ D10IOR	This overlay processes errors that occur during the Head Select-LMA Select-Read or the Head Select-LMA Select-Write disk functions. It is called by D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39).
D4LOG/ D3LOG	This overlay reports the DCU-5 Disk Error Message to the mainframe for logging in the System Log. It also displays a message on the IOP Kernel console if an unrecoverable disk error occurs. The overlay is called by D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39) at the completion of the recovery process.
D4MSG/ D3MSG	This overlay reports a message to the IOP Kernel console informing the operator of a disk error that may require manual intervention. It can be called by D4SLR or D4SKR (DD-49), or by D3SLR or D3SKR (RD-10, DD-39, and DD-40).
D4RES	The D4RES overlay performs either the clear faults or the reset disk function for DCU-5 type disk devices. It may be called by any other error recovery overlay.
D4RLR/ D3RLR	This overlay processes errors that occur on the Unit Release disk function. It is called by D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39).
D4SKR/ D3SKR	This overlay processes errors that occur on the Cylinder Select disk function. When a Cylinder Select error is detected by the interrupt handler, this overlay is called directly by D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39) to process the error. However, D4SKR/D3SKR may also be called by D4IOR/D40IOR/D3IOR/D10IOR when a Cylinder Select error is detected during the Read or Write recovery process.

Overlay	Function
D4SLR/ D3SLR	This overlay processes errors occurring on the Unit Select disk function. It is called by D4ERR (DD-49), D40ERR (DD-40), or D3ERR (RD-10 and DD-39).
D4STAT	The D4STAT overlay obtains either Drive General Status or any of the Selected Statuses the caller specifies. It may be called by any other error recovery overlay. D4STAT serves all DCU-5 error recovery activities.

3.5.2 ERROR RECOVERY PROCESS

Each major recovery process proceeds according to the following four rules:

- The clear faults and reset functions are retried a limited number of times on each call to D4RES. If the retry limit is reached with no success in performing the specified function, the error is considered unrecoverable and the entire recovery process is terminated.
- The function to obtain Drive General Status or a Selected Status is also retried a limited number of times in D4STAT before a bad status is returned to the caller. This is not considered a fatal condition, however, and the calling overlay may continue with the recovery process.
- A successful read or write process is considered to include the head select, LMA select, and read or write functions. If an error is detected on any of these functions, the process is retried starting with the head select.
- If the DN flag is not set when expected, or if the BZ and DN flags cannot be cleared with the channel clear function, the error is considered unrecoverable and the recovery process is terminated.

The following subsections describe the recovery process for each major error type.

3.5.2.1 Unit select process

The following are conditions for unit select process error recovery.

3.5.2.1.1 Software detected errors: if a software time-out has occurred, call D4RES to reset the drive and retry the select.

3.5.2.1.2 Status register 0 errors: if the drive is not ready, delay and check repeatedly until the drive becomes ready or a retry limit is reached. If an input parity error is detected, retry the select.

3.5.2.1.3 DD-49 drive general status errors: if a sequence-operation-in-progress is detected, delay and check repeatedly until the sequence operation is complete or a retry limit is reached. If a catastrophic drive error is detected, inform the operator through D4MSG that manual intervention may be required (see subsection 3.5.3, Operator Messages, for more information). If the operator indicates a retry should be performed, retry the select.

3.5.2.1.4 RD-10, DD-39, and DD-40 drive general status errors: if a catastrophic drive error is detected, inform the operator through D3MSG that manual intervention may be required (see subsection 3.5.3, Operator Messages, for more information). If the operator indicates a retry should be performed, retry the select.

If none of the above conditions are found, call D4RES to reset the drive and retry the select.

3.5.2.2 Cylinder select process

The following are conditions for cylinder select process error recovery:

3.5.2.2.1 Software detected errors: if a software time-out has occurred, call D4RES to reset the drive and retry the seek.

3.5.2.2.2 Status register 0 errors: if the drive is not ready, delay and check repeatedly until the drive becomes ready or a retry limit is reached. If an input parity error is detected, call D4RES to reset the drive and retry the seek.

3.5.2.2.3 DD-49 drive general status errors: if a sequence-operation-in-progress is detected, delay and check repeatedly until the sequence operation is complete or a retry limit is reached. If an invalid option, invalid command, function parity error, Bus-out parity error, or function lost is detected, call D4RES to clear faults. If a catastrophic drive error is detected, inform the operator through D4MSG that manual intervention may be required. If the operator indicates a retry should be performed, retry the seek.

3.5.2.2.4 RD-10, DD-39, and DD-40 drive general status errors: if a catastrophic drive error is detected, inform the operator through D3MSG that manual intervention may be required. If the operator indicates a retry should be performed, retry the seek. If it is a function parity error or Bus-out parity error, call D4RES to clear faults. If it is a command error or sequence parity error, call D4RES to reset the drive.

If none of the preceding conditions are found, call D4RES to reset the drive and retry the seek.

3.5.2.3 Head select-LMA select-read process

The following are conditions for head select-LMA select-read processing.

3.5.2.3.1 Software detected errors: if an initial LMA echo error has occurred, reload the Local Memory address into the LMA register in error until the load is successful or a retry limit is reached. If a final LMA echo error has occurred, the error is unrecoverable. If a software time-out has occurred, call D4RES to reset the drive and retry the read process.

3.5.2.3.2 Status register 0 errors: if the drive is not ready, delay and check repeatedly until the drive becomes ready or a retry limit is reached. If an input parity error is detected, retry the read process.

3.5.2.3.3 DD-49 drive general status errors: if a sequence-operation-in-progress is detected, delay and check repeatedly until the sequence operation is complete or a retry limit is reached. If an invalid option, invalid command, function parity error, Bus-out parity error, or function lost is detected, call D4RES to clear faults. If a seek error is detected, call D4SKR to perform seek error recovery. If an overflow is detected, call D4RES to clear faults. If an ID-not-found or Synchronization time-out is detected, execute retries according to the offset algorithm that follows below. If a drive error is detected, call D4SKR to perform seek error recovery. If an ECC error is detected, attempt error correction according to the correction algorithm that follows.

3.5.2.3.4 RD-10, DD-39, and DD-40 drive general status errors: if Unit Ready is not set, call D3SKR to perform seek error recovery. If it is a function parity error or Bus-out parity error, call D4RES to clear faults. If it it a command error or sequence parity error, call D4RES to reset the drive. If a seek error is detected, call D3SKR to perform seek error recovery. If an overflow is detected, call D4RES to clear faults. If an ID-not-found or Synchronization time-out is detected, execute retries according to the offset algorithm below. If a drive fault is detected, call D3SKR to perform seek error recovery. If an interface logic fault is detected, call D3SKR to perform seek error recovery. If an ECC error is detected, attempt error correction according to the following correction algorithm.

The offset algorithm is as follows:

- Call D4RES to clear faults and retry the read until a limit is reached.
- If ID-not-found error, call D4SKR (DD-49) or D3SKR (RD-10, DD-39, and DD-40) to perform seek error recovery and retry the read once more.

- Offset actuator or actuators in error toward spindle.
- Retry the read until a limit is reached.
- Offset actuator or actuators in error away from spindle.
- Retry the read until a limit is reached.

The correction algorithm is as follows:

This algorithm is superimposed on the offset algorithm. It is executed following each read retry yielding an ECC error. Compute and transfer the correction vectors for this read attempt. Compare the correction offsets from this read with those from the previous read.

If the error offsets are consistent (within 1 parcel) on all channels, call D4ECC (DD-49), D40ECC (RD-10 and DD-40), or D3ECC (DD-39) to correct the last read data.

If none of the preceding conditions are found, retry the read process.

3.5.2.4 Head select-LMA select-write process

The following are conditions for head select-LMA select-write processing:

3.5.2.4.1 Software detected errors: if an initial LMA echo error has occurred, reload the Local Memory address into the LMA register in error until the load is successful or a retry limit is reached. If a final LMA echo error has occurred, retry the write process. If a software time-out has occurred, call D4RES to reset the drive and retry the write process.

3.5.2.4.2 Status register 0 errors: if the drive is not ready, delay and check repeatedly until the drive becomes ready or a retry limit is reached. If an input parity error is detected, retry the write process.

3.5.2.4.3 DD-49 drive general status errors: if a sequence-operation-in-progress is detected, delay and check again until the sequence operation is complete or a retry limit is reached. If an invalid option, invalid command, function parity error, Bus-out parity error, or function lost is detected, call D4RES to clear the faults. If a seek error is detected, call D4SKR to perform seek error recovery. If an underflow is detected, call D4RES to clear faults. If an ID-not-found or synchronization time-out is detected, call D4RES to clear faults and retry the write process. If the retry limit is reached for ID-not-found, call D4SKR to perform seek error recovery and retry the write process one more time. If drive error is detected, call D4SKR to perform seek error recovery.

3.5.2.4.4 RD-10, DD-39, and DD-40 drive general status errors: if Unit Ready is not set, call D3SKR to perform seek error recovery. If it is a function parity error or Bus-out parity error, call D4RES to clear faults. If command error or sequencer parity error, call D4RES to reset the drive. If a seek error is detected, call D4SKR to perform seek error recovery. If an underflow is detected, call D4RES to clear faults. If an ID-not-found or Synchronization time-out is detected, call D4RES to clear faults and retry the write process. If the retry limit is reached for ID-not-found, call D3SKR to perform seek error recovery and retry the write process one more time. If drive error or interface logic error is detected, call D3SKR to perform seek error recovery.

If none of the preceding conditions are found, retry the write process.

3.5.2.5 Unit release process

The following are conditions for unit release processing.

3.5.2.5.1 Software detected errors: if a software time-out has occurred, call D4RES to reset the drive and retry the release.

3.5.2.5.2 Status register 0 errors: if the drive is not ready, delay and check repeatedly until the drive becomes ready or a retry limit is reached. If any other error is detected, call D4RES to reset the drive; reselect the unit, then retry the release.

3.5.2.5.3 DD-49 drive general status errors: if any error is detected in drive general status, call D4RES to reset the drive and end error recovery.

3.5.2.5.4 RD-10, DD-39, and DD-40 drive general status errors: if any error is detected in drive general status, end error recovery.

If none of the preceding conditions are found, reselect the unit and retry the release.

3.5.3 OPERATOR MESSAGES

If a catastrophic drive error is detected during RD-10, DD-39, DD-40, or DD-49 error recovery, a message is displayed on the IOP Kernel console informing the operator that manual intervention may be required.

The format of the message for the DD-49 is as follows:

hh:*mm*:*ss* DD49 CH *ch* FATAL *ermsg* ERROR. RETRY? ('Y' or 'N')

hh:mm:ss Time of error

ch IOP channel

ermsg Message describing the catastrophic drive status; *ermsg* is one of the following:

- BLOWER AIR
- OVERTEMP
- R/W LOGIC POWER
- RUN SWITCH
- SPINDLE POWER
- SPINDLE SPEED
- WRITE PROTECT

The format of the message for an RD-10, DD-39, or DD-40 is as follows:

hh:mm:ss DD*xx* CH *ch* UNT *un* FATAL *ermsg* ERROR. RETRY? ('Y' or 'N')

hh:mm:ss Time of error

xx Disk type:
10 RD-10
39 DD-39
40 DD-40

ch IOP channel

un Unit number

ermsg Message describing the catastrophic drive status; *ermsg* is one of the following:

- DE SEQUENCE CHECK
- UNIT READY
- WRITE PROTECT (DD-10/DD-40)
- STATUS UNAVAILABLE (DD-40)

Typing Y in response to this message causes error recovery to execute more retries. Typing N causes error recovery to terminate with an unrecoverable error status. A field engineer can perform any necessary recovery actions and indicate whether or not more retries are to be executed.

The installation parameter I@MSGRD4 defined in $APTEXT allows a site to disable disk error messages requiring a response. If disk error messages are disabled, only the information portion of the message is displayed, and error recovery immediately terminates the disk request as unrecovered.

3.5.4 ERROR REPORTING

When a DCU-5 disk error occurs, the IOS sends a disk error packet to the mainframe for logging in the System Log. The error packet contains detailed error information that can be formatted at a later time for printer output by the EXTRACT utility. For more information about EXTRACT, see the Operational Aids Reference Manual, publication SM-0044.

For the formats of the DD-49 Disk Error Packet (EM@), the DD-40 Disk Error Packet (DM@, XM@, and T@), the DD-39 Disk Error Packet (DM@), or the RD-10 Disk Error Packet (DM@, XM@, and T@) see the IOS Table Descriptions Internal Reference Manual, publication SM-0007. This error packet is created in the IOP with the disk in error. Overlay D4LOG (DD-49) or D3LOG (RD-10, DD-39, and DD-40) writes the packet to Buffer Memory and then requests ICOM in MIOP to send it to the mainframe. ICOM reads the packet in from Buffer Memory, breaks it into 6-word segments, and sends each segment over the 6-Mbyte channel. All segments for one error packet are sent with IOP system interrupts disabled, ensuring that no other MIOP-mainframe communication interrupts the sequence.

The final error status is also returned along with the successful word transfer length in the DAL that made the original request. Valid error statuses that may be returned in field DA@RC of the DAL are:

Status	Description
DAR$OK	No error encountered
DAR$REC	Recovered error
DAR$COR	Corrected data error
DAR$UNC	Uncorrected data error
DAR$UNR	Unrecovered error

If an uncorrected or unrecovered error occurs in the DD-49, D4LOG displays an error message at the IOP Kernel console in the following format:

hh:mm:ss DD49 ERROR CH *chan* CYL *cyl* HD *hd* CTL *ctl* GEN *gen type*

If an uncorrected or unrecovered error occurs in an RD-10, DD-39, or DD-40, D3LOG displays an error message with the following format:

hh:mm:ss DD*xx* ERROR CH *chan* UN *un* CYL *cyl* HD *hd* CTL *ctl* GEN *gen type*

hh:mm:ss Time of error

xx Disk type:
- 10 RD-10
- 39 DD-39
- 40 DD-40

chan IOP disk channel in error

un Unit number in error

cyl Cylinder in error

hd Head group in error

ctl Controller status

gen Drive General status

type Major error categories are as follows:

- READ Read sector process
- RLSE Unit release
- SEEK Cylinder select
- SLCT Unit select
- WRITE Write sector process

3.6 STRIPED DISK GROUPS

A striped disk group is a set of physical disk units treated logically as a single device. Requests to move data to or from such a device are broken up into pieces that are handled in parallel by the physical units in the group.

The IOS supports the configuration of one or more striped disk groups, each consisting of two to seven physical units. The maximum number of physical units is based on the track size of the unit (RD-10, DD-19, DD-29, DD-39, DD-40, or DD-49). This number is limited by the largest sector number that can fit in the request field DA@SEC. See the COS Operational Procedures Reference Manual, publication SM-0043, or the UNICOS System Administrator's Guide for CRAY Y-MP, CRAY X-MP, and CRAY-1 Computer Systems, publication SG-2018, for more information about configuring a striped disk group.

From the mainframe, a striped disk group looks like a single device with tracks containing two through seven times the number of sectors found on a single physical unit, depending on the number of units in the group.

Requests to a striped disk group are processed in the MIOP. The request packet is identical to other disk requests except that the target IOP (DA@IOP) is the MIOP. The request is mapped onto the physical units that constitutes the striped group. Individual requests are then spawned to the physical units attached to the BIOP or DIOP. The MIOP collects responses from the physical devices. When all devices have responded, a single response for the request to the striped group is returned to the mainframe.

3.6.1 LOGICAL TO PHYSICAL ADDRESS MAPPING

Each disk unit in a striped group is numbered, based on its relative position in the group, from 0 to *n*-1; *n* is the number of units in the group. The order of the units is determined by the order in which the units are specified at the time of configuration.

The logical head and cylinder numbers map 1-to-1 to the physical head and cylinder numbers, since widening the tracks of the striped group does not affect the actual number of tracks.

Figure 3-1 shows a configured striped-disk group.

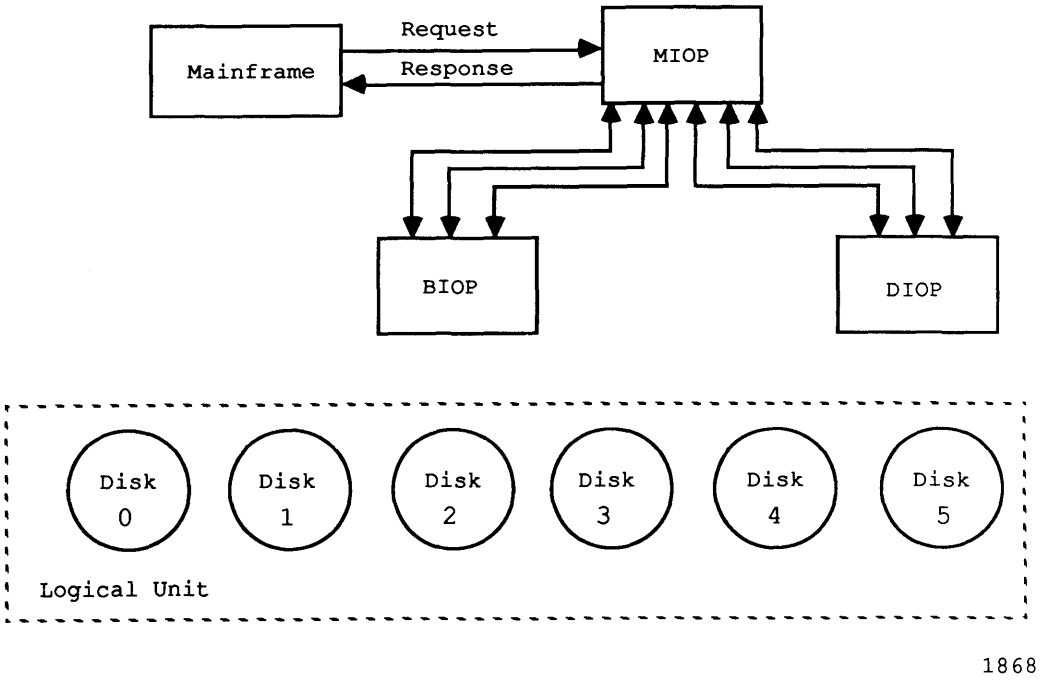


Figure 3-1. Striped Group (Six Physical Units Constituting One Logical Unit)

The logical sector number is used to determine the physical unit and the physical sector on that unit. The logical sector number is divided by the number of units in the group (logical sector/number of units = physical sector + physical unit number).

The quotient is the physical sector:

The physical sector is from 0 to *secs*-1; *secs* is the number of physical sectors per track on a unit.

The remainder is the physical unit number:

The physical unit number is from 0 to *n*-1; *n* is the number of units in the group.

In summary, all logical sectors that reside on a physical unit are equivalent MOD *n*; *n* is the number of units in the group.

Example (a three-unit striped group):

Logical Sectors	Physical Sector	Unit
37=3 x 12 + 1	12	1
38=3 x 12 + 2	12	2
39=3 x 13 + 0	13	0
40=3 x 13 + 1	13	1
41=3 x 13 + 2	13	2
42=3 x 14 + 0	14	0

3.6.2 STEPFLOW FOR A REQUEST TO A STRIPED GROUP

A request to a striped group is received by the MIOP, then handled by BIOP or DIOP, and finally, dispatched by the MIOP, as follows:

- MIOP

  When the MIOP receives a request, the CDEM demon overlay becomes active, as follows:

  1. A request packet is received with the destination ID (DA@DID) specifying disk (RQ$DISK). The request is passed to the CDEM overlay for processing.

  2. CDEM detects that the target IOP for the request (DA@IOP) is the MIOP. CDEM locates the table for the device using the logical channel number (DA@CHN) as an index into the look-up table DCCB. These tables are built at initialization time using the configuration information in AMAP.

  3. CDEM maps the logical request into physical requests to the units in the group and sends the requests to the appropriate IOPs for processing.

- BIOP/DIOP

  BIOP or DIOP request processing for striped disk groups essentially means determining the target memory address that corresponds to a disk sector for a request.

  4. Requests received for I/O to a unit within a striped group are handled the same as for I/O to an individual disk unit. The only distinguishable difference is in the value contained in the request field DA@UNS. This field specifies the number of units in the group for which this unit is a member. The value in DA@UNS is used to map the target memory (DA@TM0/TM1) to sequential sectors of the request (DA@SEC). The target memory address (M_1) that corresponds to a disk sector for a request is determined according to the following algorithm:

     $$(S_1 - S_0) \times D'512 \times U + M_0 = M_1$$

     The difference between the current sector (S_1) and the base sector DA@SEC (S_0) is multiplied times the sector size (D'512), which is then multiplied by the number of units in the group DA@UNS (U); and finally, this number is added to the target memory base address DA@TM0/TM1 (M_0).

     For a single device, DA@UNS contains a 1, which maps sequential sectors in memory to sequential sectors on disk, as figure 3-2 shows. Figure 3-3 shows memory mapping for a two-unit group.

  5. A response is sent to the MIOP for each unit as I/O completes.

- MIOP

  The last steps in the flow for striped disk request processing involve sending a single request response to the mainframe.

  6. Responses received for each unit are in the demon overlay ACOM for DCU-4 type disk units, or ICOM for DCU-5 type disk units. Based on the value in field LCH being nonzero, the responses are passed on to the AMSG overlay for processing.

  7. When I/O is complete, AMSG collects the responses from each unit in the group. When all have responded, a single response is returned to the mainframe.

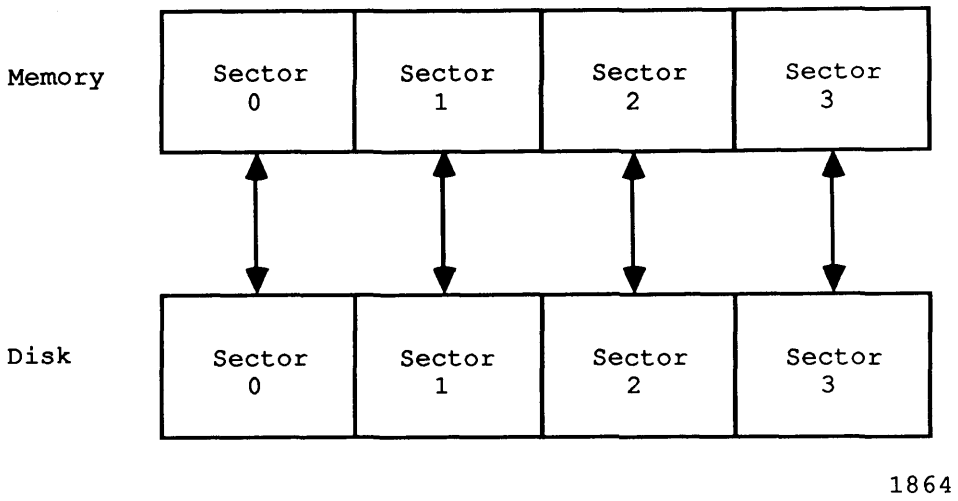


Figure 3-2. Target Memory Mapping for a Single Device

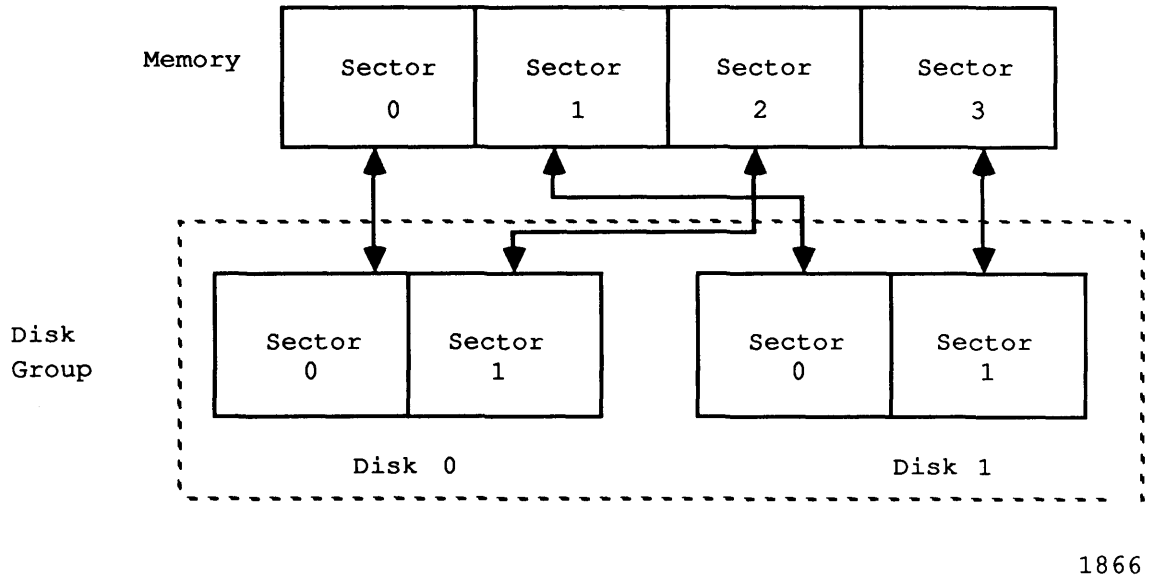


Figure 3-3. Target Memory Mapping for a Two-unit Group

3.7 KERNEL INTERNAL DISK I/O

Some Kernel routines must transfer data to or from disk. (For example, the DKDMP routine dumps disk data directly to the IOS Peripheral Expander printer.) Two Kernel overlays, DISKIO and DKIOEX, furnish these routines with the ability to reference disk space through normal disk protocol.

The overlay that requires disk I/O calls the DISKIO overlay and specifies the necessary parameters. If the I/O occurs in a different IOP, DISKIO uses the AWAKE and ALERT function requests to activate overlay DKIOEX in the relevant IOP. DISKIO then waits for the response that signals completion of the request.

The following sequence in the caller's overlay calls the internal disk I/O mechanism.

Location	Result	Operand
	CALL	DISKIO,(RD|WRT,*iop,chnl,unit,cyl,head,sector,length,mosu,mosl*)

RD
WRT — Direction of disk I/O:

RD = 1 Read from disk
WRT = 2 Write to disk

iop — Number of the IOP to perform the I/O; 1 through 3 are legal.

chnl — Channel number; 20 through 37_8 are legal.

unit — Unit number; 0 through 2 are legal for DD-39 disks and 0 through 1 are legal for DD-40 disks. The parameter should be 0 for all other device types.

cyl — Cylinder number

head — Head number

sector — Sector number

length — Length (in words) of data to be transferred

mosu — High-order bits of Buffer Memory address into which or from which data is to be transferred

mosl — Low-order bits of Buffer Memory address into which or from which data is to be transferred

DISKIO returns one of the following statuses to the caller in the A register:

Status	Description
0	Normal completion
1	Bad parameter supplied by caller
2	Bad disk status; unable to complete I/O.
3	Resources unavailable; I/O not done.

4. TAPE EXEC

The Tape Exec software (TEX) is composed of activities necessary to accomplish the following:

- Route messages between I/O processors
- Process mainframe requests
- Format and move tape data
- Recover from hardware and software errors

4.1 ARCHITECTURE

The Tape Exec portion of the tape subsystem executes in the Master I/O processor (MIOP), the Buffer I/O processor (BIOP), and the Auxiliary I/O processor (XIOP).

The MIOP is responsible for routing tape requests and responses between the mainframe tape software driver and the XIOP.

The BIOP is responsible for moving tape data between Central Memory and Buffer Memory over the 100-Mbyte data channel. Tape Exec routines in the BIOP perform their function based on requests from the XIOP. Each request from the XIOP to the BIOP refers to a Data Stream Control (DSC) table in Buffer Memory. A DSC is used to hold data that is in transit between Central Memory and a tape device. See the I/O Subsystem (IOS) Table Descriptions Internal Reference Manual, publication SM-0007, for details of the DSC.

The XIOP is responsible for processing tape requests received from the mainframe software driver and generating appropriate responses. Tape Exec software uses the Block Multiplexer Channel (BMX) subsystem to issue physical device commands and manage tables associated with the tape subsystem. See section 5 for a description of the BMX subsystem. The XIOP uses the Tape Exec software in BIOP to move data to and from Central Memory in the mainframe. It communicates with the BIOP via request packets and shared DSC tables in Buffer Memory. Tape Exec software in XIOP also provides error recovery routines for handling software or hardware errors encountered by the tape subsystem.

4.1.1 TAPE EXEC ACTIVITY

Each tape device is controlled by a unique Tape Exec activity (TEX) created when the device is opened. Each TEX activity executes in the XIOP and controls most of the request processing for the device. A TEX activity terminates when a close request has been processed for its associated tape device. Each TEX activity allocates a data structure in Local Memory called a Tape Control Block (TCB) associated with the device. The TCB holds information about the device and contains queues for communicating with other activities in the tape subsystem. See I/O Subsystem (IOS) Table Descriptions Internal Reference Manual, publication SM-0007, for details of the TCB.

4.1.2 BYPASS ACTIVITY

The request interface with the mainframe driver allows for two separate types of I/O to be active simultaneously. The mainframe may request that data be moved between Central Memory and Buffer Memory in 512-word sector units. At the same time, the mainframe may request that data be moved between Buffer Memory and the tape device in block size units. The mainframe specifies the block size on each request. This overlap of the two types of data movement allows the mainframe to use the 100-Mbyte data channel of the BIOP for fast access to buffered data, while the XIOP handles movement to and from the slower tape devices. The mainframe attempts to stay ahead of user requests for data by building a read-ahead area in Buffer Memory for each tape device being read. In a similar fashion, the mainframe attempts to off-load data to a write-behind data area in Buffer Memory for each device being written. The mainframe driver controls the size of each of these areas by the frequency and size of data transfer and block I/O requests.

The BYPASS activity in the XIOP has primary responsibility for handling I/O requests from the mainframe. It executes as a single activity and processes I/O requests for all devices. BYPASS examines each request for data transfer or block I/O. If sectors of data are to be transferred between Central Memory and Buffer Memory, BYPASS calls the data transfer activity in the BIOP. If block I/O is requested, BYPASS activates the appropriate TEX activity, if not already active. The BYPASS activity handles Buffer Memory allocation for data transfer and block I/O by calling the BUFMAN routine.

4.1.3 DATA STREAM CONTROL TABLE

The read-ahead and write-behind data areas in Buffer Memory are controlled by a DSC for each active device. A DSC is created in Buffer Memory at device open time by the BMXTPO routine calling the DSCGET routine. Each DSC table is a Buffer-Memory-resident data structure shared between the various components of the tape subsystem executing in the BIOP and the XIOP. The mainframe interface allows the stream of requests for user data to be interrupted for processing of label data, or user job special end of volume processing. When this occurs, the primary data stream is held and a secondary DSC is allocated for the new data stream. The BYPASS activity controls the stacking and popping of primary and secondary DSC tables by calling the DSCGET routine for each request requiring a change of data stream. The mainframe request interface also allows for discarding data in either the primary or secondary data stream. The DSC table is retained when discarding of data is requested. All DSC tables for a device are deallocated by TEX calling DSCGET at device close time.

A DSC table consists of a header area and a number of buffer descriptor entries. The header area is divided into two sections, one for the XIOP parameters used for moving data between Buffer Memory and the device, the other for the BIOP parameters used for moving data between the Buffer and Central Memories. In general, each IOP references only its own section of the DSC header. The XIOP tape software references its section of the DSC header so often that a copy is kept in Local Memory in each TCB. This minimizes the number of Buffer Memory reads and writes needed to maintain the XIOP header section.

The BIOP does not need to reference its section of the DSC header as often, so it uses Buffer Memory I/O. Local Memory is also much scarcer in the BIOP due to the need for a large number of disk buffers. This prevents allocating any TCB type tables for device information storage.

When a DSC is allocated, the DSCGET routine computes the number of 64-bit buffer descriptor entries needed by using the installation maximum block size parameter. Each descriptor entry can describe a 512-word sector of data. A DSC is allocated to hold the header plus enough descriptor entries for at least two maximum sized data blocks. The DSC buffer descriptor entries are used in a circular fashion to identify the start and length of Buffer Memory data blocks. Pointers are kept in the XIOP DSC header that demark the range of active buffer descriptor entries. A limit value is also kept in the DSC header to describe the physical size of the DSC table. The BUFMAN routine in the XIOP has primary responsibility for maintaining the DSC pointers and circular list of descriptor entries.

4.1.4 TDEM1 ACTIVITY

The TDEM1 activity in the BIOP has primary responsibility for moving data between the Buffer and Central Memories. It executes as a single activity and processes data transfer requests for all devices. TDEM1 receives requests from the BYPASS activity and the TAPEIO routine of each of the TEX device activities in the XIOP. Requests to TDEM1 contain the number of 512-word sectors of data to be transferred along with the Buffer Memory address of the DSC table for the requested device. TDEM1 supports three data formats used by the mainframe: Transparent format, Interchange format, and List I/O format.

Transparent data format is used to transfer blocks of data between the Central and Buffer Memories based on the block size specified in the request, without any internal or external control word structures. Transparent format is used by both the COS and UNICOS operating systems.

Interchange format is used only by COS to allow common library and system I/O routines for tape and disk data. The data format uses an internal control word structure to mark the end of tape blocks. Each control word is 64 bits. Control words are added to the data by TDEM1 as data is transferred from Buffer to Central Memory. Control words are removed when data is transferred from Central to Buffer Memory.

List I/O format is used exclusively by UNICOS to transfer blocks of tape data between the Central and Buffer Memories. Block length is communicated in a list structure external to the actual data. TDEM1 builds this list structure and passes it to the mainframe on each read data transfer. TDEM1 reads and decodes the contents of the list structure on each write data transfer.

4.1.5 TAPE ERROR RECOVERY ACTIVITIES

Each TEX device activity initiates error recovery by calling the TAPERR routine. TAPERR determines the type of device in error and creates an appropriate error recovery activity. The TCART overlay is the highest level routine in the error recovery activity for cartridge devices. The TERROR overlay is called for noncartridge devices. A new error recovery activity is created each time a new error is encountered while attempting recovery of an earlier error. This mechanism prevents overflow of the Kernel SMOD structure associated with an activity in situations where multiple errors are present.

Error conditions may include software generated errors, BMX channel errors, and device/control unit errors. The error recovery activity attempts recovery and reports ending status to the calling TEX device activity, or calling error recovery activity. An error message is formatted and displayed on the XIOP Kernel console. An error response packet is generated and sent to the mainframe to be included in the system log file.

4.2 REQUEST AND RESPONSE PACKET ROUTING

Tape request packets are six words in length. Each request packet is received from the mainframe by the MIOP over the 6-Mbyte low-speed channel. All tape response packets are sent to the mainframe by the MIOP over the 6-Mbyte low-speed channel. Each request packet contains a source and destination ID field (TQ@SID, TQ@DID). The source ID for request packets reflects the mainframe ID, normally C1. The destination ID field contains either an ASCII G or D. Initial configuration information is passed to the MIOP in a request G-packet. All other tape requests contain a D in the destination ID field. Response packets to the mainframe reverse the values of source and destination IDs.

The CDEM routine in the MIOP and BCOM routines (BCOM0, BCOM1, and BCOM3) handle interprocessor routing of tape packets. CDEM and the BCOM routines use the Kernel A-to-A message passing software for communicating between processors via the MBAQ, MBBQ, and MBDQ tables in the respective Kernels.

The CDEM routine in the MIOP passes requests from the mainframe to the XIOP. BCOM0 in the MIOP passes responses from XIOP back to the mainframe. It handles D-packet responses as well as error response E-packets to be sent to the mainframe system log. BCOM0 ensures that multiple packet error responses are sent in consecutive order to the mainframe.

BCOM1 executes in the BIOP to handle requests from XIOP for data transfer over the 100-Mbyte channel. The actual data transfer is done by the TDEM1 routine. BCOM1 queues A-to-A messages directly to TDEM1 using the BXQQ table in the BIOP Kernel.

BCOM3 executes in the XIOP to handle requests from MIOP and responses from BIOP. Requests from MIOP are routed to the appropriate portion of Tape Exec software for processing. Responses from BIOP are routed to the mainframe via the MIOP.

Most of the Tape Exec routines send response packets directly to the MIOP for routing to the mainframe.

4.3 REQUEST PROCESSING

BCOM3 handles initiation of processing for tape request packets received from the mainframe. Each tape request packet contains an ordinal (TQ@DVN) by which the associated device is known to the mainframe and the tape subsystem. Device ordinals start at 0 and are unique for each device. The XDEVMAX entry in the XIOP Kernel table area specifies the number of devices configured. Each tape request packet contains a function code (TQ@FCN) along with parameters needed for processing the specific request.

4.3.1 CONFIGURATION CHANGE REQUEST (FC$CHNGE)

Configuration change requests are processed by BCOM3 creating the CONMAN routine. CONMAN calls the BMX routines BMXCPU or BMXCON to perform the actual device functions and configuration changes to the tape subsystem tables. Pointers to the channel table list (XCHT), the device table list (XDEV), the control unit bank tables list (XCBT), and the device bank tables list (XDBT) are in the XIOP Kernel table.

BCOM3 recognizes the FC$CHNGE request and creates CONMAN activity to process it. If the activity cannot be created, a protocol error response is returned to the mainframe.

CONMAN examines the packet for type of configuration change (TQ@TYP). If it is the initial mainframe request to configure the entire tape subsystem, CONMAN does a Goto BMXCPU. A response is not sent to the mainframe on the initial configuration request.

If the request is for an individual component of the subsystem (channel, control unit, or device), CONMAN validates the request parameters. Only one component may be changed per request. A protocol error will be returned to the mainframe if more than one component (TQ@CHN, TQ@CNT, TQ@DEV) is specified. A channel or control unit may be configured on-line or off-line (TQ@OPC). A device may be configured on-line or off-line (TQ@NAV) and up or down (TQ@OPC). CONMAN calls BMXCON to make the requested change. BMXCON returns the status of the request to CONMAN, which sends a response to the mainframe. (Section 5 describes the BMXCON and BMXCPU routines.)

Figure 4-1 shows the processing of configuration change requests.

4.3.2 MOUNT REQUEST (FC$MOUNT)

Mount requests are handled by BCOM3 creating the BMXOPE routine to perform the open for a tape device. BMXOPE does a Goto to BMXTPO for the actual mount processing. BMXTPO in turn does a Goto to the Tape Exec routine TEX to become the device activity for the mounted drive.

Figure 4-2 shows the processing of mount requests.

BCOM3 - Validates the requested device ordinal (TQ@DVN) to see if it is in range (0 to XDEVMAX-1). If not, a protocol error response is sent to the mainframe.

BCOM3 checks the Device Table Open flag (BDV@OP) to see if a TEX activity exists for the requested device. If not open, the BMXOPE activity is created to process the request. If the activity cannot be created, a protocol error response is sent to the mainframe.

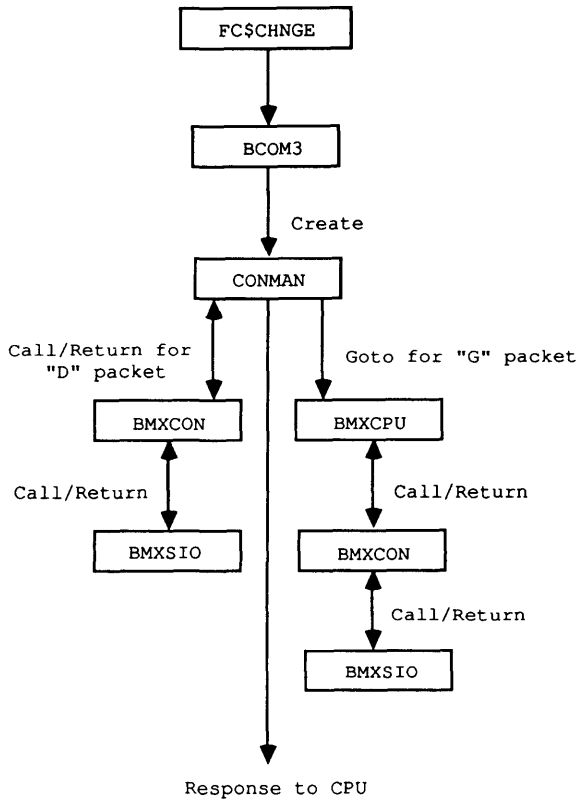


Figure 4-1. Processing of Configuration Change Requests

BMXOPE - When the requested device has not been opened, BCOM3 always assumes the requested function is FC$MOUNT. If it is not, BMXOPE will return a protocol error to the mainframe. BMXOPE checks to see if a device activity currently owns the device (BDV@AI). This can occur if a configuration change is taking place on the device. If a device activity exists, BMXOPE waits for it to release the device, and then assigns itself as the device activity (BMXOPE will eventually become the TEX activity). BMXOPE marks the device open (BDV@OP) and does a Goto to the BMXTPO routine for mount processing.

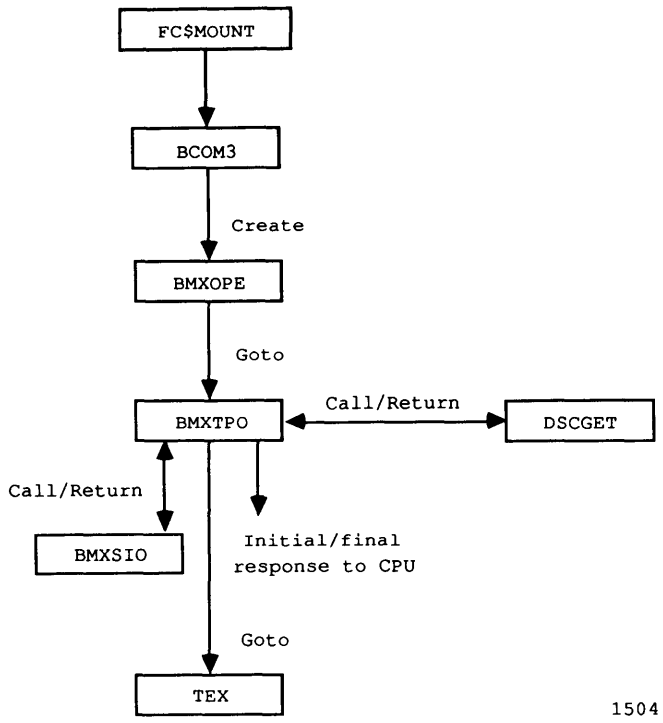


Figure 4-2. Processing of Mount Requests

BMXTPO - Allocates control tables for the TEX activity. A TCB table is allocated (DC@) which contains the Command Parameter Block (CPB@) used to interface to the BMX I/O subsystem. A DSC table (CU@, NX@, BF@) is allocated in Buffer Memory by a call to DSCGET.

BMXTPO arms the drive for load point. If the drive is not ready with a mounted tape at load point, an initial response indicating the not ready status is returned to the mainframe (ST@RDY). When the drive is ready with a tape at load point, final status is sent to the mainframe (ST@BOT) along with any appropriate write protect status (ST@NRW). BMXTPO does a Goto to TEX.

TEX - Waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.3 READ REQUEST (FC$READ)

Read I/O requests are passed by BCOM3 to the BYPASS activity for processing through the DATQU queue in the XIOP Kernel. BYPASS handles initiation of any data transfer by sending a request to TDEM1 in the BIOP. The BYPASS activity queues requests for tape blocks to be read to the appropriate TEX activity for the requested device. BYPASS handles stacking of the user and label DSC tables by calling the DSCGET routine. BYPASS handles Buffer Memory allocation for the user and label DSC tables by calling the BUFMAN routine.

Figure 4-3 shows the processing of read requests.

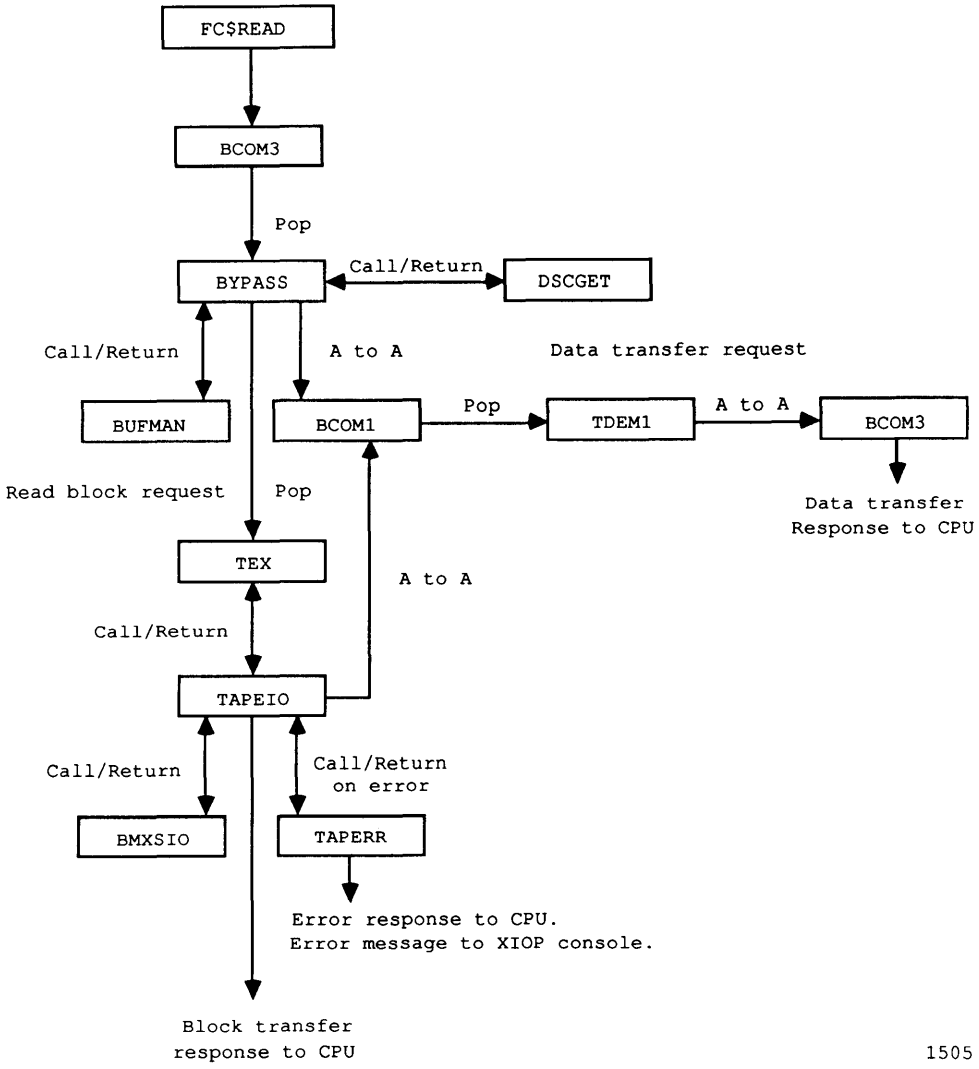


Figure 4-3. Processing of Read Requests

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the BYPASS activity on the DATQU in the Kernel. BYPASS is activated, if waiting, by popping the TIMQU in the Kernel.

BYPASS - Dequeues the next request from DATQU and locates the TCB for the requested device. The Hold Data flag (DD@HLD) in the request is examined. If set, and the user DSC table has not been saved, DSCGET is called. If the Hold Data flag is not set and the user DSC table is being held (DC@DHU, DC@DHL), DSCGET is called to restore the stacked DSC table (DC@DSU, DC@DSL).

A read request from the mainframe may include a request for sectors of data (TQ@RSC) to be transferred from Buffer Memory to Central Memory, a request for data blocks (TQ@RBC) to be read from tape, or both.

BYPASS assumes that the mainframe has only one request for data sectors outstanding at a time to the IOS.

The transfer of a data sector may be truncated by the mainframe setting the Partial Sector flag in the request (DD@PCW). The number of words to transfer (TQ@PWC) indicates that the mainframe buffer is less than a full 512-word sector. This mechanism is typically used for reading label data. BYPASS ensures that a single sector of data is being requested. Otherwise, a protocol error is returned.

BYPASS validates list parameters if the data format is List I/O.

The number of sectors of data requested is compared to the number of sectors of read-ahead stored in Buffer Memory. The sectors of read-ahead (CU@VMS) is kept in the Local Memory copy of the DSC header (DC@DSC) in the TCB. If any read-ahead sectors are present, a request is sent to TDEM1 in BIOP to transfer the lesser of the request size and read-ahead size. The Central Memory address (DA@HSU, DA@HSL) and address of the DSC (DA@DSU, DA@DSL) are included in the request.

If the request size is larger than the read-ahead size, the excess requested sectors are saved in the TCB (DC@RSC) and will be processed by TAPEIO as blocks are read from tape. The Central Memory address is also adjusted and saved (DC@HSU, DC@HSL).

BYPASS (continued) - The mainframe request may also contain a requested block count to cause blocks to be read from tape. The interface to the mainframe allows multiple block requests to be outstanding. This creates a synchronization problem when the IOS encounters an error or tape mark while reading. The mechanism used to get the mainframe and the IOS back in sync is the Next Valid Packet flag (DD@NVP) in the request. This signals the IOS that the mainframe has received a previous error or tape mark response and wishes to resume processing. Read block requests received between the time of the error and receipt of the request with next valid packet set are discarded by BYPASS. Requests for transfer of data sectors are not affected by this mechanism.

BYPASS calls BUFMAN to allocate Buffer Memory for the requested number of blocks.

BUFMAN - The mainframe directs Buffer Memory allocation by the IOS based on the total available Buffer Memory size parameter returned in all responses (TQ@MOS). It uses this value to determine how many new blocks may be requested for each active device.

BUFMAN allocates read-ahead buffers for all devices. The DSC structure contains a header area and descriptors for each Buffer Memory buffer allocated. The window of active buffers is described by a top (CU@TOP) and bottom (CU@BTM) pointer in the XIOP section of the DSC header. These pointers are the word offset in the DSC of the descriptor for the first buffer in the window and the next to be allocated, respectively. Buffers are allocated at the bottom of the window and deallocated from the top. A limit value (CU@LIM) identifies the physical end of the descriptor list in the DSC. The DSC may be multiple 512-word sectors. The DSCGET routine allocates the needed size at mount time based on the IOS installation maximum block size parameter (MBS$MAX). MBS$MAX is used to create a DSC large enough to hold a minimum of two blocks of data plus the DSC header.

Before allocating buffers for the request, BUFMAN deallocates buffers that have been transferred to the mainframe. The top and BIOP (NX@PTR) pointers mark this range of descriptors. After deallocation, The top pointer is adjusted to equal the BIOP pointer. Deallocation does not clear descriptors in the Buffer Memory DSC in order to minimize Buffer Memory accesses. Excess Buffer Memory accesses by BUFMAN could delay the processing of tape channel interrupts and cause software overrun errors.

BUFMAN (continued) - BUFMAN computes the number of buffers to allocate based on the number of blocks requested and the actual maximum block size in the request (TQ@MBU, TQ@MBL). Buffers previously allocated at the bottom of the list that are not in use are subtracted from the allocation count. This range is marked by the XIOP (CU@PTR) and bottom pointers. Not-in-use buffers result from the actual tape block size being less than the maximum block size specified in previous requests. A calculation is performed to determine if descriptors for the allocation count will fit in the DSC. If not, an error code (1) is returned to BYPASS. If the descriptors will fit, allocation is performed.

BUFMAN checks for high priority activities waiting to execute before starting allocation. It yields the processor to any such waiting activities in order to allow I/O to proceed on active devices. This minimizes the possibility of software overrun errors.

Allocation of buffers is done by calls to the Kernel GETDISK routine. If buffer space is not available, an error code (2) is returned to BYPASS, and any buffers allocated are released. The maximum allocation count is requested on the first call to the GETDISK routine. GETDISK may return fewer than the number requested. By repeated requests the allocation count will eventually be satisfied. Buffers allocated on each request to GETDISK are at contiguous addresses in Buffer Memory. BUFMAN builds a descriptor for each 512-word buffer allocated in its Local Memory copy of the DSC. Each descriptor contains a count (BF@CTG) of the number of buffers with increasing address that are contiguous. This value is used extensively by the TDEM and TDEM1 routines to minimize Buffer Memory accesses for descriptors and data. Portions of the DSC are moved to and from Local Memory using a paging mechanism to minimize Buffer Memory accesses.

BYPASS - Sends a protocol error response to the mainframe if Buffer Memory is not available to satisfy the request or the DSC cannot hold enough descriptors for the increased read-ahead area.

Block I/O is sustained by the outstanding block count in the TCB (DC@RBC) being nonzero. BYPASS adds the new requested block count to this field. If the TEX activity is not executing, BYPASS will activate it by popping DC@QUA and placing the mainframe request on the DC@MSG queue.

BYPASS (continued) - BYPASS uses the I/O Active Control flag (DC@IOF) to determine if the TAPEIO routine of the TEX activity is still checking for new requested blocks to read (DC@RBC). BYPASS considers the device inactive if the control flag is not set and activates the TEX activity. If TEX is executing, the added block count will sustain I/O.

BYPASS processes all requests queued to it by BCOM3 and waits on TIMQU when finished.

At this point, a data transfer or block I/O may be in progress for each active mainframe read request.

BCOM1 - Data transfer requests are received in BIOP via the A-to-A message mechanism of the Kernel. BCOM1 receives messages from XIOP sent by BYPASS or TAPEIO. It queues the A-to-A message to the TDEM1 routine on BXQQ queue for processing. TDEM1 is activated, if waiting, by BCOM1 popping the TDMQ queue in the Kernel. BCOM1 does not read the request from Buffer Memory to minimize the number of packet areas in use in BIOP.

TDEM1 - TDEM1 handles all data transfer to and from Central Memory in the mainframe for the tape subsystem. It allocates a static area in Local Memory for the request and the DSC header. It dynamically allocates a Local Memory buffer for each request processed. TDEM1 begins by reading the request from Buffer Memory sent by XIOP and the shared DSC header structure.

TDEM1 checks the data format (DD@FMT) in the request. The three formats supported include: Transparent (FM$TRNS), List I/O (FM$LIST), and Interchange (FM$BLKD). The appropriate read subroutine is called for processing.

TDEM1 read subroutine processing is driven by the requested sector count specified by the XIOP. The requested sector count is always less than or equal to the number of data sectors in the read-ahead area. Data sectors in the read-ahead area are always part of completed tape blocks. The BIOP (NX@PTR) and XIOP (CU@PTR) DSC header pointers define the range of the read-ahead data area. As a result, the BIOP pointer always trails the XIOP pointer in the circular descriptor list for reads. Each read routine uses a common set of routines for reading and updating DSC descriptor entries.

TDEM1 (continued) - TDEM1 processes requests to move data between the Buffer and Central Memory by interpreting the contents of the buffer descriptor entries built by XIOP for each tape block. Each descriptor entry contains a status field (BF@STA). A value of MD$BOR indicates that the descriptor refers to the first sector of a tape data block. Subsequent sectors in the block have a zero in the status field of their descriptors. Special values of MD$EOV, MD$EOF, and MD$EOD in the status field are used by the Interchange data format read routine discussed below. Each descriptor entry also contains a count of the number of contiguous Buffer Memory buffers that follow the buffer referred to by the descriptor (BF@CTG). TDEM1 attempts to minimize the Buffer Memory I/O involved in reading and updating descriptor entries by using this field to predict the contents of successive descriptor entries. In this way, the address field (BF@ADU, BF@ADL) of successive entries can be produced in Local Memory without reading the Buffer Memory DSC. Each descriptor with a status field value of MD$BOR contains the length of the associated tape block in bytes (BF@RLU, BF@RLL). All other descriptor entries contain zero in the length field.

TDEM1 also attempts to use the bypass data transfer hardware between Buffer and Central Memory, available on the IOS Model C, for transfers larger than one sector. The current transfer limit is eight sectors to prevent tying up the high speed and Buffer Memory channels. This limit prevents impact on disk and front-end data traffic. TDEM1 will yield the processor to high priority activities after each data transfer to allow prompt servicing of disk channels and transfer of front-end data.

The Transparent format read routine begins by assuming that the maximum limit of eight sectors can be transferred to Central Memory. If the descriptor for the next sector to be moved (NX@SCT) indicates the number of contiguous data sectors is less than eight, the contiguous count becomes the maximum transfer size. If the next sector to be moved is part of a block of data with an unrecovered read error (BF@DBF), the maximum transfer size is limited to a single sector. The Partial Sector flag (DD@PCW) in the request is checked next. If set, the transfer is limited to the number of words specified in the request (DA@PWC); otherwise, full sectors are assumed. The current size is compared with the remaining number of sectors in the Central Memory buffer. The smaller of these two becomes the transfer size. Finally, the transfer size is compared to the number of bytes left in the current tape block. The smaller of these two values is used as the final transfer

TDEM1 (continued) - size. After the transfer completes, the next sector to be moved pointer (NX@SCT) is incremented. If the transfer completed the current tape block, the BIOP block pointer (NX@PTR) is updated. Finally, the Central Memory buffer address is updated in anticipation of the next transfer. The above sequence is repeated until the requested sector count has been satisfied, or a sector of bad data has been transferred. A transfer of bad data causes the flag (DA@DBF) to be set in the response packet sent to BCOM3 in the XIOP.

Transparent read processing completes by computing the number of words transferred in the last sector (DA@PWC), and the number of unused bits in the last word transferred (DA@UBC). Both values are returned in the response. The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are also returned, along with the data transfer status bit (ST@DTR), in the response.

The List I/O format read routine attempts to transfer complete tape blocks to Central Memory (DA@HSU, DA@HSL). The mainframe supplies the address in Central Memory of the list structure (DA@LSU, DA@LSL). The list is a table of 64-bit words where each word is used to describe one tape block (TL@). The list size (DA@LSS) may be from 1 to 512 words.

Each time a mainframe data transfer request is received, a new list address is supplied. As blocks of data are moved to Central Memory to satisfy the request, successive entries in the list are filled in by TDEM1. TDEM1 keeps a pointer to the next position in the list (NX@LPT) in the DSC header. TDEM1 cannot tell when an entire request has been processed because it can receive a request from BYPASS and multiple requests from TAPEIO for the same mainframe request. BYPASS and TAPEIO set the New List Received flag (DA@LSR) for the first request to TDEM1 of each new mainframe request. This enables TDEM1 to know when to reset NX@LPT to the beginning of a new list.

The processing loop begins by comparing the number of bytes (BF@RLU, BF@RLL) in the next data block to be moved to the space remaining in the Central Memory buffer. If the next tape block will not fit, the transfer request is terminated and the status (ST@LSE) is set in the response packet. This implies that the Central Memory buffer must be large enough (DA@MBU, DA@MBL) to hold at least one tape block.

TDEM1 (continued) - If the next data block will fit in the Central Memory buffer, the list entry describing the block is built. The status (TL@FMT) and block length (TL@BCU, TL@BCL) are set in the entry. If the block to be transferred contains unrecovered data, the Bad Data flag (TL$DBF) is also set in the list entry and in the response packet (DA@DBF). The transfer request terminates after moving a bad data block.

The data transfer portion of the List I/O processing loop begins by assuming the limit of eight sectors can be moved to Central Memory. If the descriptor for the next sector to be moved (NX@SCT) indicates the number of contiguous data sectors is less than eight, the contiguous count becomes the current transfer size. This value is compared to the number of bytes left in the tape block to be moved. The smaller of these two values is used as the final transfer size. After the transfer completes, the Next Sector To Be Moved pointer (NX@SCT) is incremented. The Central Memory buffer address is updated in anticipation of the next transfer. The BIOP pointer (NX@PTR) is updated when the entire tape block has been transferred. The loop above is repeated until the next tape block does not fit in the remaining Central Memory buffer area, or until a block of bad data has been transferred.

The List I/O format read routine completes by terminating the list structure with a zero entry, unless the list size limit (DA@LSS) has been reached. The list is written to Central Memory (DA@LSU, DA@LSL) and the list pointer (NX@PTR) is updated in the BIOP DSC header. The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are returned, along with the data transfer status bit (ST@DTR), in the response to BCOM3.

The Interchange format read routine converts tape data to COS Interchange format when transferring data to Central Memory by inserting control words into each sector moved.

Each sector of data begins with a block control word (BCW). Tape blocks are terminated with a record control word (RCW). Files and datasets are terminated with end-of-file and end-of-data control words (RCW), respectively. See I/O Subsystem (IOS) Table Descriptions Internal Reference Manual, publication SM-0007, for the description of block and record control words.

The insertion of control words in each sector of data limits the maximum transfer size per Kernel TRANSFER request to one sector. Each sector must be constructed in a Local Memory buffer before being sent to Central Memory.

TDEM1 (continued) - The transfer loop begins by building a BCW in the first word of the Local Memory buffer. Each sector contains just one BCW. This allows a maximum of 511 words of data and control words to be included in each sector.

The sector is filled based on the status field (BF@STA) of the current (NX@PTR) DSC buffer descriptor entry.

A value of MD$BOR or MD$EOV indicates that part of a data block is to be moved to the sector in the Local Memory buffer. If all data in the tape block has been moved, an end of record RCW is generated at the next position in the sector.

If the status value is MD$EOV, and the sector is not full, the Null flag (CW@NUL) is set in the RCW to indicate that the remainder of the sector is empty. The descriptors containing the MD$EOV values are modified by the TAPEND routine in XIOP when a mainframe FC$EORR request is processed. The mainframe uses this request to read a partial sector of data at end-of-volume processing.

An MD$EOF descriptor status indicates that an end of file RCW is to be added to the Local Memory buffer. If the sector is not full, the Null flag (CW@NUL) is set to indicate that the remainder of the sector is empty. The descriptor entry containing the MD$EOF value is built by the TAPEND routine in XIOP when a mainframe FC$EOFR request is processed. The mainframe uses the FC$EOFR request to insert control words into the data stream corresponding to embedded tape marks read in a multiple file tape dataset.

The MD$EOD status indicates that an end-of-file RCW and an end-of-data RCW are to be added to the Local Memory buffer. If both control words will not fit in the current sector, the NX@MOD value in the BIOP DSC header is set to indicate that only the end-of-file control word was generated. The next request for data transfer will cause TDEM1 to recognize that an end-of-data control word is still needed. If the sector is not full after both control words have been generated, the Null flag (CW@NUL) is set to indicate that the remainder of the sector is empty. The descriptor containing the MD$EOD value is built by the TAPEND routine in XIOP when a mainframe FC$EODR request is processed. The mainframe uses the FC$EODR request to terminate user and label data streams with the end-of-file and end-of-data control words.

TDEM1 (continued) - Each tape block is moved to the Local Memory buffer following the end of record RCW for the previous block. The Buffer Memory address of the data is computed from the current sector pointer (NX@SCT) and word offset (NX@WRD) in the sector. The length to be moved may include data from the next sector of Buffer Memory. Because the next sector of data might not be physically contiguous in Buffer Memory, two partial moves may be required. The contiguous count field (BF@CTG) in the descriptor is used to determine the number of moves required. The smaller of the contiguous data, the remaining data in the tape block, and the remaining space in the Local Memory buffer is moved.

The forward word index (CW@FWI) in the control word preceding the data just moved or control word just generated is updated. The forward word index links BCW and RCW control words for the library routines in the mainframe. The Bad Data flag (CW@DBF) is also set in the preceding control word, if the data just moved was part of a block containing unrecovered tape data. The Bad Data flag allows the library routines in the mainframe to skip bad tape blocks, if requested by the user job.

The above loop continues, adding data and control words, until the sector is complete. Before the sector is sent to Central Memory (DA@HSU, DA@HSL), the Partial Sector Transfer flag is examined in the packet (DD@PCW). If set, the transfer size is limited to DA@PCW words, else the full sector is transferred on the 100-Mbyte data channel from Local to Central Memory.

Each sector requested by XIOP is constructed and transferred by the above loop, until requested sector count has been satisfied, or a sector of bad data has been transferred. A transfer of bad data causes the flag (DA@DBF) to be set in the response packet sent to BCOM3 in the XIOP. The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are also returned, along with the data transfer status bit (ST@DTR), in the response.

The BIOP section of the Buffer Memory DSC is updated with current values for the device when the the response is sent to BCOM3 in the XIOP. TDEM1 continues processing new requests from BXQQ queue and waits on TDMQ queue when finished.

BCOM3 - BCOM3 in the XIOP receives the read data transfer response packet from TDEM1. It subtracts the transferred sector count (DA@TSC) from the count of read-ahead data sectors in Buffer Memory (CU@VMS). The updated total of data sectors is placed in the response packet (TQ@VMS). The number of unallocated sectors of Buffer Memory is computed and also placed in the packet (TQ@MOS). The data transfer response is sent to the mainframe through the MIOP.

BCOM3 checks for error conditions (no ST@DTR, TQ@DBF, ST@LSE) that terminate any additional sector transfers for the request. If an error occurred, the residual sector count (DC@RSC) in the TCB is cleared to prevent TAPEIO from initiating any new requests to TDEM1 in the BIOP.

Finally, BCOM3 activates any activities waiting for the data transfer response by popping the BIOP wait queue (DC@QUB) in the TCB and decrementing the count of outstanding requests to BIOP (DC@BRQ).

This concludes the description of the read data transfer processing initiated by BYPASS.

TEX - The TEX activity is activated by BYPASS when all previous read block requests from the mainframe have been satisfied. TEX dequeues the new request from DC@MSG.

If an error terminated a previous read block request, the Next Valid Packet flag (DD@NVP) is checked in the request. The mainframe signals that it received the previous error status by setting the Next Valid Packet flag to resume processing. The error flag (DC@ERR) is cleared when next valid packet is recognized by TEX. Read block request packets received without Next Valid Packet flag set are ignored by TEX when the error flag is set in the TCB.

TEX allocates two Local Memory buffers for reading tape data and saves their addresses in the TCB (DC@BFA, DC@BFB). If two buffers are not available, TEX allocates none and waits on the TXBQU queue in the Kernel. Other activities releasing Local Memory buffers will Pop this queue.

TEX calls the TAPEIO routine to initiate read processing for the device.

TAPEIO - Provides the data I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPEIO (continued) - TAPEIO processes read block I/O requests until the outstanding block count (DC@RBC) in the TCB is satisfied, or an error occurs.

TAPEIO attempts to build multiple read commands for the BMX subsystem when possible. The chaining of commands allows the BMX driver to sustain data I/O transfer at the rate of the device. Each command in the chain represents a request to read one block from the device. TAPEIO limits command chains to ten blocks in order to allow other devices to access the BMX channels and control units. This allows a fair distribution of I/O among active devices, without a significant loss in transfer rate on any particular device.

Each command to the BMX subsystem is stored in a Channel Program Word (CPW@) structure in the CPB. TAPEIO uses three CPW structures in a circular fashion for command chaining. Each CPW contains a flag (CPW@CC) to indicate whether the command is chained to the next command. TAPEIO builds a CPW for each block to be read (up to three) and calls BMXSIO to initiate the read command chain.

TAPEIO checks the operation status (CPB@OS) and count of commands complete (CPB@CD) when BMXSIO returns. Normally, BMXSIO returns one command complete for each call by TAPEIO. If the data blocks are small, or activity on other devices is heavy, more than one command may complete before BMXSIO can return to TAPEIO. The Return Kernel service function can allow another activity to gain control of the processor, which can delay the return to TAPEIO. In this case, the operation status (OS$) applies to the last command of the count completed.

TAPEIO processes each tape block just read. For each block, the descriptor in the DSC referred to by the current block pointer (CU@PTR) is updated with MD$BOR status (BF@STA) and the block length (BF@RLU, BF@RLL) in bytes from the CPW for the block read. The completed CPW is rebuilt if a command chain is active. The current block pointer (CU@PTR) is advanced to refer to the descriptor for the next block to be read. The outstanding block request count (DC@RBC) in the TCB is decremented. The count of sectors of data in the Buffer Memory read-ahead area is incremented by the size of the block just read. If the data format is Interchange, the number of control words to be added by TDEM1 will be included in the calculation. If a request for transfer of read-ahead sectors (DC@RSC) is present in the TCB, a request will be generated and sent to TDEM1 in BIOP for the number of sectors now available.

TAPEIO (continued) - This allows data to move to the mainframe as quickly as possible. Finally, a block finished response is generated with status (ST@BTR) and sent to the mainframe through the MIOP. The number of data sectors in the read-ahead area (TQ@VMS) and the total number of Buffer Memory sectors available for allocation (TQ@MOS) are included in the response.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPEIO rebuilds the command chain and calls BMXSIO to initiate I/O.

An operation status of OS$BZ implies that the command chain is continuing. TAPEIO calls BMXSIO to wait for the next command to complete.

A status of OS$DN implies that all commands in the chain are done. TAPEIO checks for new block requests added by BYPASS (DC@RBC). If a requested block count is present, TAPEIO builds a new CPW list and initiates I/O by calling BMXSIO again. If no new block requests have been received, TAPEIO clears the I/O Active flag (DC@IOF) and returns to TEX.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active command chain in progress. TAPEIO clears the I/O Active flag (DC@IOF) and returns to TEX.

Finally, an operation status of OS$ER indicates that the BMX subsystem detected an error on the last command in the count completed. The error may be related to a hardware condition (channel error, unit check, unit exception, mid-block CCR), or a software detected condition (overrun, large block).

TAPEIO detects the unit exception condition encountered by examining the Tape Mark Read flag (CPB@TE). An end-of-file status (ST@EOF) is returned, along with block finished status (ST@BTR), in the mainframe response for the command. The End-of-file Detected flag (DC@EOF) is set in the TCB. TAPEIO clears the I/O Active flag and returns to TEX. TEX will wait for the next mainframe request with Next Valid Packet flag (DD@NVP) set before processing is resumed.

TAPEIO (continued) - If no other errors are present, TAPEIO checks for a large block error by examining the Length Error flag (CPB@LE). The large block condition is detected by the BMX subsystem when a tape block larger than the specified maximum block size (TQ@MBU, TQ@MBL) is read. Data beyond the block size limit is discarded. TAPEIO responds with the large block status (ST@BIG), along with block finished (ST@BTR), in the mainframe response for the command. TAPEIO clears the I/O Active flag and returns to TEX. TEX will wait for the next mainframe request with Next Valid Packet flag (DD@NVP) set before processing is resumed.

Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the Device Detected error flag (CPB@DD) is set. If either type of error is present, TAPEIO calls TAPERR to create an error recovery activity to retry the failed command.

TAPERR returns the status of the recovery attempt. A status of zero indicates the command was recovered successfully. Normal end-of-command processing is performed, and I/O continues.

TAPERR may also return an unrecovered data check status (ST@URE) to indicate that the data in the tape block could not be read. The tape is still positioned properly, as if the block had been read. In this case, TAPEIO will set the Bad Data flag (BF@DBF) in all descriptor entries in the DSC for the failed block. If no data was recovered, a descriptor entry is still reserved for the empty block and a byte length of one (BF@RLU, BF@RLL) is set to prevent the software from dealing with blocks of zero length. TAPEIO performs normal end-of-command processing and I/O continues.

Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPEIO clears the I/O Active flag and returns to TEX. TEX will wait for the next mainframe request with Next Valid Packet flag (DD@NVP) set before processing is resumed.

TEX - The return from TAPEIO causes TEX to release the two Local Memory buffers (DC@BFA, DC@BFB). If an error response was returned, the TCB error flag (DC@ERR) is set. The error flag invokes the next valid packet mechanism for checking mainframe requests. If an error is noted, the BUFMAN routine is called to deallocate Buffer Memory buffers for requested blocks that were not used by TAPEIO.

BUFMAN - BUFMAN is called to deallocate unneeded read buffers from the bottom (CU@BTM) of the circular descriptor list. The area between the XIOP block pointer (CU@PTR) and the bottom of the list defines the range of buffers to release. After deallocation, the bottom pointer is adjusted to equal the XIOP block pointer. This is the only instance where a descriptor entry pointer moves in a backward direction.

Descriptor entries between the top pointer (CU@TOP) and new bottom pointer may still contain references to the buffers just deallocated in their contiguous count fields (BF@CTG). The BF@CTG field is adjusted in all active descriptor entries in the list to reflect the deallocation.

TEX - Waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.4 WRITE REQUEST (FC$WRITE)

Write I/O requests are passed by BCOM3 to the BYPASS activity for processing through the DATQU queue in the XIOP Kernel. BYPASS handles initiation of any data transfer by sending a request to TDEM1 in the BIOP. The BYPASS activity queues requests for tape blocks to be written to the appropriate TEX activity for the requested device. BYPASS handles stacking of the user and label DSC tables by calling the DSCGET routine. BYPASS handles Buffer Memory allocation for the user and label DSC tables by calling the BUFMAN routine.

Figure 4-4 shows the processing of a write request.

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the BYPASS activity on the DATQU in the Kernel. BYPASS is activated, if waiting, by popping the TIMQU in the Kernel.

BYPASS - Dequeues the next request from DATQU and locates the TCB for the requested device. The Hold Data flag (DD@HLD) in the request is examined. If the Hold Data flag is set and the user DSC table has not been saved, then DSCGET is called. If the Hold Data flag is not set and the user DSC table is being held (DC@DHU, DC@DHL), DSCGET is called to restore the stacked DSC table (DC@DSU, DC@DSL).

A write request from the mainframe may include a request for sectors of data (TQ@RSC) to be transferred from Central Memory to Buffer Memory, a request for data blocks (TQ@RBC) to be written to tape, or both.

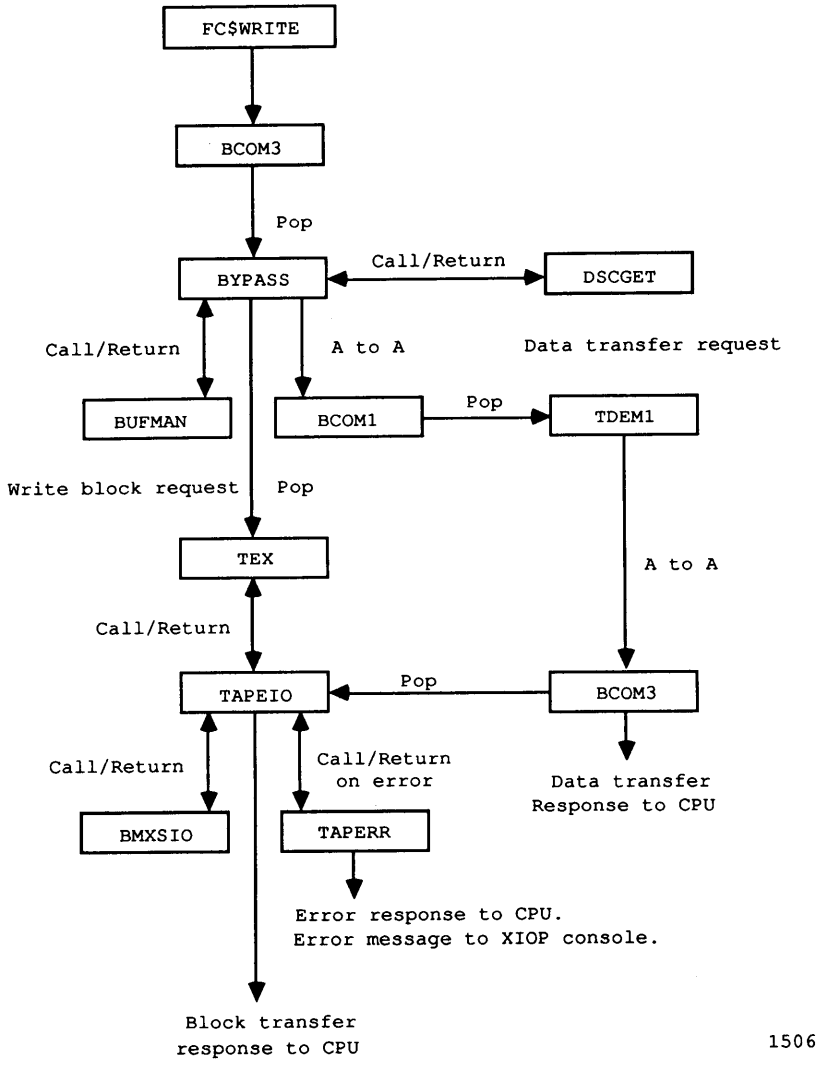


Figure 4-4. Processing of Write Requests

BYPASS (continued) - BYPASS assumes that the mainframe has only one outstanding request at a time to the IOS for data sectors.

BYPASS validates list parameters if the data format is List I/O.

The mainframe can request that data sectors (TQ@RSC) be transferred to the Buffer Memory write-behind area. Any tape blocks ending in these data sectors must be indicated in the request (TQ@RBC). The absence of a block request count indicates that the data sectors all belong to the current block being assembled in the write-behind area.

BYPASS (continued) - BYPASS attempts to minimize the number of calls to BUFMAN for allocation by only making requests as new blocks are transferred from the mainframe. This presents a problem on the first data transfer request when the write-behind area is empty. BYPASS solves the problem by making an extra call to BUFMAN in this situation to allocate an extra buffer of maximum block size (TQ@MBU, TQ@MBL). This extra buffer allows BYPASS to always stay one buffer ahead of mainframe data transfer requests.

BYPASS must also make a special call to BUFMAN to adjust the size of the extra buffer when the mainframe increases the maximum block size on a request without a block count. BYPASS detects this increase by saving the maximum block size from each data transfer request in the TCB (DC@MBU, DC@MBL). The saved value is compared to the block size in each new request to see if the mainframe has increased it. BUFMAN adjusts the size of the extra buffer based on the new block size value.

BYPASS calls BUFMAN to allocate Buffer Memory for the requested number of blocks.

BUFMAN - The mainframe directs Buffer Memory allocation by the IOS based on the total available Buffer Memory size parameter returned in all responses (TQ@MOS). It uses this value to determine how many new blocks may be requested for each active device.

BUFMAN allocates write-behind buffers for all devices. The DSC structure contains a header area and descriptors for each Buffer Memory buffer allocated. The window of active buffers is described by a top (CU@TOP) and bottom (CU@BTM) pointer in the XIOP section of the DSC header. These pointers are the word offset in the DSC of the descriptor for the first buffer in the window (CU@TOP) and the next buffer to be allocated (CU@BTM). Buffers are allocated at the bottom of the window and deallocated at the top. A limit value (CU@LIM) identifies the physical end of the descriptor list in the DSC. The DSC may be multiple 512-word sectors. The DSCGET routine allocates the needed size at mount time based on the IOS installation maximum block size parameter (MBS$MAX). MBS$MAX is used to create a DSC large enough to hold a minimum of two blocks of data plus the DSC header.

BUFMAN (continued) - Before allocating buffers for the request, BUFMAN deallocates buffers that have been written to tape by TAPEIO. The top and XIOP (CU@PTR) pointers mark this range of descriptors. After deallocation, the top pointer is adjusted to equal the XIOP pointer. To minimize Buffer Memory accesses, deallocation does not clear descriptors in the Buffer Memory DSC. Excess Buffer Memory accesses by BUFMAN could delay the processing of tape channel interrupts and cause software overrun errors.

BUFMAN computes the number of buffers to allocate based on the number of blocks requested and the actual maximum block size in the request (TQ@MBU, TQ@MBL). Buffers previously allocated at the bottom of the list that are not in use are subtracted from the allocation count. This range is marked by the BIOP (NX@PTR) and bottom pointers. Not in use buffers result from the actual tape block size being less than the maximum block size specified in previous requests. A calculation is performed to determine if descriptors for the allocation count will fit in the DSC. If not, an error code (1) is returned to BYPASS. If the descriptors will fit, allocation is performed.

BUFMAN checks for high priority activities waiting to execute before starting allocation. It yields the processor to any such activities in order to allow I/O to proceed on active devices. This minimizes the possibility of software overrun errors.

Allocation of buffers is done by calls to the Kernel GETDISK routine. If buffer space is not available, an error code (2) is returned to BYPASS, and any buffers allocated are released. The maximum allocation count is requested on the first call to the GETDISK routine. GETDISK may return fewer than the number requested. By repeated requests, the allocation count will eventually be satisfied. Buffers allocated on each request to GETDISK are at contiguous addresses in Buffer Memory. BUFMAN builds a descriptor for each 512-word buffer allocated in its Local Memory copy of the DSC. Each descriptor contains a count (BF@CTG) of the number of buffers with increasing addresses that are contiguous. This value is used extensively by the TDEM and TDEM1 routines to minimize Buffer Memory accesses for descriptors and data. Portions of the DSC are moved to and from Local Memory using a paging mechanism to minimize Buffer Memory accesses.

BYPASS - BYPASS sends a protocol error response to the mainframe if Buffer Memory is not available to satisfy the request or if the DSC cannot hold enough descriptors for the increased write-behind area.

BYPASS (continued) - BYPASS checks the mainframe request for the special Last Block Write flag (DD@LBW) used for Transparent format data. The last block write request is used to force a data block in the write-behind area that is less than maximum block size to be considered a full block. No data transfer takes place for the request in TDEM1 in the BIOP.

BYPASS generates a request to TDEM1 in the BIOP to move the requested data sectors to the write-behind area, or process the Last Block Write flag, if set.

Another special flag in the mainframe request indicates whether blocks moved to the write-behind area are to be written to tape. If set, the No Write flag (DD@NWR) causes BYPASS not to activate the TEX activity.

Block I/O is sustained while the count of blocks (CU@VMS) in the write-behind area is nonzero. BCOM3 adds to this field on data transfer responses from TDEM1 in BIOP as blocks are transferred to the write-behind area.

BYPASS will activate the TEX activity by popping DC@QUA and placing the mainframe request on the DC@MSG queue. BYPASS uses the I/O Active Control flag (DC@IOF) to determine if the TAPEIO routine of the TEX activity is still checking for new requested blocks to write (CU@VMS). BYPASS considers the device inactive if the control flag is not set, and activates the TEX activity. If TEX is executing, the added block count will sustain I/O.

BYPASS processes all requests queued to it by BCOM3 and waits on TIMQU when finished.

At this point, a data transfer or block I/O may be in progress for each active mainframe write request.

BCOM1 - Data transfer requests are received in BIOP through the A-to-A message mechanism of the Kernel. BCOM1 receives messages from XIOP sent by BYPASS or TAPEIO. It queues the A-to-A message to the TDEM1 routine on BXQQ queue for processing. TDEM1 is activated, if waiting, by BCOM1 popping the TDMQ queue in the Kernel. BCOM1 does not read the request from Buffer Memory to minimize the number of packet areas in use in BIOP.

TDEM1 - TDEM1 handles all data transfer to and from Central Memory in the mainframe for the tape subsystem. It allocates a static area in Local Memory for the request and the DSC header. It dynamically allocates a Local Memory buffer for each request processed. TDEM1 begins by reading the request from Buffer Memory sent by XIOP and the shared DSC header structure.

TDEM1 (continued) - TDEM1 checks the data format (DD@FMT) in the request. The three formats supported include: Transparent (FM$TRNS), List I/O (FM$LIST), and Interchange (FM$BLKD). The appropriate write subroutine is called for processing.

TDEM1 write subroutine processing is driven by the requested sector count specified by the XIOP. The write-behind area in Buffer Memory is described by the BIOP (NX@PTR) and XIOP (CU@PTR) DSC header pointers. As a result, the XIOP pointer always trails the BIOP pointer in the circular descriptor list for writes. Each write routine uses a common set of routines for reading and updating DSC descriptor entries.

TDEM1 processes requests to move data between the Central and Buffer Memories by generating buffer descriptor entries for each tape block moved.

Each descriptor entry contains a status field (BF@STA). A value of MD$BOR indicates that the descriptor refers to the first sector of a tape data block. Subsequent sectors in the block have a zero in the status field of their descriptors. A special value of MD$EOF in the status field is used by the Interchange and List I/O data formats to indicate an end-of-file mark is to be written to tape.

Each descriptor entry also contains a count of the number of contiguous Buffer Memory buffers that follow the buffer referred to by the descriptor (BF@CTG). TDEM1 attempts to minimize the Buffer Memory I/O involved in reading and updating descriptor entries by using this field to predict the contents of successive descriptor entries. In this way, the address field (BF@ADU, BF@ADL) of successive entries can be produced in Local Memory without reading the Buffer Memory DSC. Each descriptor with a status field value of MD$BOR contains the length of the associated tape block in bytes (BF@RLU, BF@RLL). All other descriptor entries contain zero in the length field.

TDEM1 also attempts to use the bypass data transfer hardware between Buffer and Central Memory (available on the Model C IOS) for transfers larger than one sector. The current transfer limit is eight sectors to prevent tying up the high-speed and Buffer Memory channels. This limit prevents impact on disk and front-end data traffic. TDEM1 will yield the processor to high priority activities after each data transfer to allow prompt servicing of disk channels and transfer of front-end data.

TDEM1 (continued) - The Transparent format write routine begins by checking that the length of the current block being assembled is within maximum block size (DA@MBU, DA@MBL). If the block size has been exceeded, the Large Block Status flag (ST@BIG) is set in the response to BCOM3 in XIOP.

The data transfer loop begins by assuming that the maximum limit of eight sectors can be transferred to Buffer Memory. If the descriptor for the next sector to be filled (NX@SCT) indicates the number of contiguous buffers is less than eight, the contiguous count becomes the current transfer size. The current size is compared with the remaining number of data sectors in the Central Memory buffer. The transfer size becomes the smaller of the two. Finally, the transfer size is compared to the number of bytes needed to fill the current tape block. The smaller of these values is used as the final transfer size. After the transfer completes, the next sector to be filled pointer (NX@SCT) is incremented. If the transfer completed the current tape block, the descriptor for the first sector of the block is marked with MD$BOR status and the block length in bytes (BF@RLU, BF@RLL). The BIOP block pointer (NX@PTR) is updated. Finally, the Central Memory buffer address is adjusted in anticipation of the next transfer. This sequence is repeated until all requested sectors have been moved.

Transparent write processing completes by checking for the Last Block Write flag (DD@LBW) set in the request. If set, any partial tape block is marked with the MD$BOR status and the number of bytes in the block (BF@RLU, BF@RLL). The BIOP block pointer (NX@PTR) is adjusted. The request may also contain the Sync Request flag (DD@SNC). This causes any partial data block to be discarded, if set. The BIOP sector pointer (NX@SCT) is reset to the beginning of the block. The word offset pointer (NX@WRD) and the current record length are reset to 0.

The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are returned in the response to BCOM3. If no errors are being reported, the data transferred status bit (ST@DTR) is also set.

The List I/O format write routine attempts to transfer complete tape blocks from Central Memory (DA@HSU, DA@HSL). The mainframe supplies the address in Central Memory of the list structure (DA@LSU, DA@LSL). The list is a table of 64-bit words where each word is used to describe one tape block (TL@). The list size (DA@LSS) may be from 1 to 512 words. Each time a mainframe data transfer request is received, a new list address is supplied.

TDEM1 (continued) - The List I/O write routine begins by reading the list into a Local Memory buffer from Central Memory. Successive entries in the list are decoded by TDEM1 as blocks of data are moved to Buffer Memory to satisfy the request.

The processing loop begins by examining the status field (TL@FMT) of the list entry.

A value of TL$EOR indicates a data block is to be moved to Buffer Memory. If the block length (TL@BCU, TL@BCL) is 0, a write format error status (ST@WFE) is returned to BCOM3 in the XIOP.

The block length is then validated against the maximum block size (DA@MBU, DA@MBL) specified in the request. If the maximum is exceeded, a large block error status (ST@BIG) is returned to BCOM3 in the XIOP.

The List I/O transfer loop begins by assuming the limit of eight sectors can be transferred to Buffer Memory. If the descriptor for the next sector to be filled (NX@SCT) indicates the number of contiguous data sectors is less than eight, the contiguous count becomes the current transfer size. The current transfer size is compared to the number of bytes left to be moved in the tape block. The smaller of these two values is used as the final transfer size. After the transfer completes, the next sector to be filled pointer (NX@SCT) is incremented. The Central Memory buffer address is updated in anticipation of the next transfer. The descriptor for the beginning of the block is marked with MD$BOR status and the block length field (BF@RLU, BF@RLL). The BIOP block pointer (NX@PTR) is updated when the entire tape block has been transferred.

If the list entry status is TL$EOF, the descriptor for the beginning of block is marked with MD$EOF status, the block length field is set to 0, and the BIOP block pointer is updated. If the list entry status is neither TL$EOR or TL$EOF, the write format error status (ST@WFE) is set in the response packet returned to BCOM3 in XIOP.

The loop above is repeated until all requested sectors have been transferred to Buffer Memory as complete blocks.

The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are returned in the response to BCOM3. If no errors are being reported, the data transferred status bit (ST@DTR) is also set.

TDEM1 (continued) - The Interchange format write routine converts COS Interchange format to raw tape blocks when transferring data to Buffer Memory by removing control words from each sector moved.

Each sector of data begins with a block control word (BCW). Tape blocks are terminated with a record control word (RCW). Files and datasets are terminated with end-of-file and end-of-data control words (RCW), respectively. See the I/O Subsystem (IOS) Table Descriptions Internal Reference Manual, publication SM-0007, for the description of block and record control words.

The appearance of control words in each sector of data limits the maximum transfer size per Kernel TRANSFER request to one sector. Each sector must be deblocked in a Local Memory buffer before being sent to Buffer Memory.

The transfer loop begins by reading the next sector from the Central Memory buffer into Local Memory. The first word of the sector is validated as a block control word (BCW). If a BCW (CW@MOD) is not present, the write format error status (ST@WFE) is returned to BCOM3 in XIOP.

The forward word index (CW@FWI) in the BCW describes the number of words of data between the BCW and next control word.

If data is present, the EOF Pending flag (NX@EFP) is checked in the DSC header. When an end-of-file record control word (RCW) is encountered during the deblocking of a sector of data, its interpretation is not clear until it is known what follows it in the data stream. An end-of-file RCW followed by data or another EOF RCW should be treated as a tape block. However, an EOF RCW followed by an end-of-data RCW is not considered a tape block. It just marks the end of the dataset. At times, this ambiguity cannot be resolved until the next sector of data is transferred from Central Memory. This occurs when the EOF RCW is the last word of data in a sector. In the worst case, the next sector is not available until the next mainframe data transfer request is made. The NX@EFP flag in the DSC header allows TDEM1 to remember that an EOF RCW was encountered and that its interpretation is pending. If the pending EOF RCW is followed by data, the descriptor entry for the current block in Buffer Memory is marked with the MD$EOF status. The BIOP block pointer (NX@PTR) is updated, and the NX@EFP flag is cleared.

TDEM1 (continued) - The data is next moved from the Local Memory buffer to Buffer Memory at location NX@SCT, NX@WRD. The contiguous count field (BF@CTG) in the descriptor for the sector determines whether one or two moves are needed.

If the entire Local Memory sector has not been deblocked, the next word to be examined must be a control word. If not a control word (CW@MOD), the Write Format Error Status flag (ST@WFE) is set in the response packet to BCOM3 in XIOP.

If the control word is an end-of-record control word, a nonzero length tape block should have just been moved to Buffer Memory, else a Write Format Error Status (ST@WFE) is returned to BCOM3.

The number of unused bits in the last word of the block is checked by examining the count (CW@UBC) in the control word. For tape data, the count must always be a byte multiple, else the Write Format Error Status flag (ST@WFE) is returned to BCOM3.

The number of unused bytes in the last word is subtracted from the length of the current block. The result is validated against the maximum block size (DA@MBU, DA@MBL) specified in the request. If the maximum size is exceeded, the large block error status (ST@BIG) is returned to BCOM3.

If the block length is valid, the descriptor entry for the beginning of the block in Buffer Memory is marked with MD$BOR status and the block length (BF@RLU, BF@RLL). The BIOP block pointer (NX@PTR) is updated.

If the control word is an EOF RCW, a check is made to ensure that a data block has not just been moved to Buffer Memory. Data blocks must be terminated with an end-of-record control word. If the end-of-record control word is missing, a write format error status (ST@WFE) is returned to BCOM3.

If the control word is a legitimate end-of-file, the NX@EFP flag is checked to see if the previous control word was also an EOF RCW. If so, the previous EOF RCW is treated as a tape block and marked in the Buffer Memory descriptor as such, and the NX@EFP flag is set to indicate that interpretation of the new EOF RCW is pending.

If the control word is an end-of-data RCW, the NX@EFP flag from the DSC header is checked. It should be set, because datasets are terminated with both end-of-file and end-of-data control words. If not set, the Write Format Error Status flag (ST@WFE) is returned to BCOM3.

TDEM1 (continued) - The above loop continues until the requested sector count is moved from Central Memory to Buffer Memory, an error is encountered, or an EOD RCW is processed.

Finally, the Sync Request flag (DD@SNC) is checked in the request. If set, this flag causes any partial block to be discarded. The BIOP sector pointer (NX@SCT) is reset to the beginning of block and the word offset pointer (NX@WRD) and the current record length are reset to zero.

The number of sectors (DA@TSC) and number of blocks (DA@TBC) transferred are returned in the response to BCOM3. If no errors are being reported, the data transferred status bit (ST@DTR) is also set.

The BIOP section of the Buffer Memory DSC is updated with current values for the device when the the response is sent to BCOM3 in the XIOP. TDEM1 continues processing new requests from BXQQ queue and waits for TDMQ queue when finished.

BCOM3 - BCOM3 in the XIOP receives the write data transfer response packet from TDEM1. It adds the transferred block count (DA@TBC) to the count of write-behind data blocks stored in Buffer Memory (CU@VMS). The updated total of data blocks is placed in the response packet (TQ@VMS). The number of unallocated sectors of Buffer Memory is computed and also placed in the packet (TQ@MOS). The data transfer response is sent to the mainframe through the MIOP.

BCOM3 activates any activities waiting for the data transfer response by popping the BIOP wait queue (DC@QUB) in the TCB and decrementing the count of outstanding requests to BIOP (DC@BRQ).

This concludes the description of the write data transfer processing initiated by BYPASS.

TEX - The TEX activity is activated by BYPASS when all previous write block requests from the mainframe have been satisfied. TEX dequeues the new request from DC@MSG. If an error terminated a previous write block request, the Next Valid Packet flag (DD@NVP) is checked in the request. The mainframe signals that it received the previous error status by setting the Next Valid Packet flag to resume processing. The error flag (DC@ERR) is cleared when next valid packet is recognized by TEX. Write block request packets received without Next Valid Packet flag set are ignored by TEX when the error flag is set in the TCB.

TEX (continued) - TEX allocates two Local Memory buffers for writing tape data and saves their addresses in the TCB (DC@BFA, DC@BFB). If two buffers are not available, TEX allocates none and waits for the TXBQU queue in the Kernel. Other activities releasing Local Memory buffers will Pop this queue.

TEX calls the TAPEIO routine to initiate write processing for the device.

TAPEIO - TAPEIO provides the data I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPEIO processes write block I/O requests until the outstanding block count (CU@VMS) in the DSC is satisfied or an error occurs.

TAPEIO attempts to build multiple write commands for the BMX subsystem when possible. The chaining of commands allows the BMX driver to sustain data I/O transfer at the rate of the device. Each command in the chain represents a request to write one block to the device. TAPEIO limits command chains to ten blocks in order to allow other devices to access the BMX channels and control units. This allows a fair distribution of I/O among active devices, without a significant loss in transfer rate on any particular device.

Each command to the BMX subsystem is stored in a Channel Program Word (CPW@) structure in the CPB. TAPEIO uses three CPW structures in a circular fashion for command chaining. Each CPW contains a flag (CPW@CC) to indicate whether or not the command is chained to the next command.

TAPEIO builds a CPW for each block (up to three) to be written by examining the descriptor entries for the next three blocks in Buffer Memory. The location of the descriptor for the first block (CU@PTR) is known. The location of the descriptors for the next two blocks is computed from the block length (BF@RLU, BF@RLL) of prior entries. The status (BF@STA) in each descriptor is checked for an end-of-file (MD$EOF). If present, TAPEIO will write to tape any blocks preceding the EOF, and then generate a response to the mainframe with the End-of-file Status flag (ST@EOF) set. TAPEIO will return to TEX. This allows the mainframe to request that a tape mark be written.

TAPEIO (continued) - The length of each block is set in each CPW (CPW@BU, CPW@BL). TAPEIO activates the TDEM activity to preload the first sector of the first block in the chain to Local Memory. TAPEIO calls BMXSIO to initiate the write command chain.

TAPEIO checks the operation status (CPB@OS) and count of commands complete (CPB@CD) when BMXSIO returns. Normally, BMXSIO returns one command complete for each call by TAPEIO. If the data blocks are small or activity on other devices is heavy, more than one command may complete before BMXSIO can return to TAPEIO. The Return Kernel service function can allow another activity to gain control of the processor, which can delay the return to TAPEIO. In this case, the operation status (OS$) applies to the last command of the count complete.

TAPEIO processes each tape block just written. The XIOP block pointer (CU@PTR) is updated to point to the descriptor for the next block to be written to tape. If a command chain is active, the completed CPW is rebuilt. The outstanding block request count (CU@VMS) in the DSC is decremented. A block finished response is generated with status (ST@BTR) and sent to the mainframe via the MIOP. The number of data blocks in the write-behind area (TQ@VMS) and the total number of Buffer Memory sectors available for allocation (TQ@MOS) are included in the response.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPEIO rebuilds the command chain and calls BMXSIO to initiate I/O.

An operation status of OS$BZ implies that the command chain is continuing. TAPEIO calls BMXSIO to wait for the next command to complete.

A status of OS$DN implies that all commands in the chain are done. TAPEIO checks for new block requests added by BCOM3 (CU@VMS). If the write-behind area is empty but a block is currently being moved to Buffer Memory by BIOP, TAPEIO waits for the move to complete by pushing on DC@QUB in the TCB. TAPEIO builds a new CPW list and initiates I/O by calling BMXSIO again. If no new block requests have been received, and none are in process, TAPEIO clears the I/O Active flag (DC@IOF) and returns to TEX.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active

TAPEIO - (continued) command chain in progress. TAPEIO clears the I/O Active flag (DC@IOF) and returns to TEX.

TAPEIO detects the unit exception condition encountered by examining the End-of-tape flag (CPB@TE). If the flag is set, TAPEIO will backspace over the last block written to leave room for end-of-volume labels. When processing label data, the Block Finished flag (ST@BTR) is set along with the End-of-tape flag (ST@EOT). If user data is being written to tape, just the End-of-tape flag (ST@EOT) is set in the mainframe response for the command. TAPEIO clears the I/O Active flag and returns to TEX. TEX will wait for the next mainframe request with Next Valid Packet flag (DD@NVP) set before processing is resumed.

Finally, an operation status of OS$ER indicates that the BMX subsystem detected an error on the last command in count complete. The error may be related to a hardware condition (channel error, unit check, unit exception, mid-block ccr) or a software detected condition (overrun).

Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the Device Detected error flag (CPB@DD) is set. If either type of error is present, TAPEIO calls TAPERR to create an error recovery activity to retry the failed command.

TAPERR returns the status of the recovery attempt. A zero status indicates the command was recovered successfully. Normal end-of-command processing is performed, and I/O continues.

TAPERR may also return an end-of-tape encountered status (ST@EOT). The above procedure for unit exception processing is invoked.

Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPEIO clears the I/O Active flag and returns to TEX. TEX will wait for the next mainframe request with Next Valid Packet flag (DD@NVP) set before processing is resumed.

TEX - The return from TAPEIO causes TEX to release the two Local Memory buffers (DC@BFA, DC@BFB). If an error response was returned, the TCB error flag (DC@ERR) is set. The error flag invokes the next valid packet mechanism for checking mainframe requests.

TEX waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.5 END READ REQUESTS (FC$EOFR, FC$EORR, FC$EODR)

End read requests are issued by the mainframe tape driver to terminate Interchange format data with appropriate end-of-file, end-of-record, or end-of-data control words.

The end-of-file read request (FC$EOFR) is issued when a tape mark has been encountered that is not followed by a label on a multiple file tape dataset. An end-of-file record control word will be generated by TDEM1 to complete any partial sector of data residing in the Buffer Memory read-ahead area.

The end-of-record read request (FC$EORR) is issued during special end-of-volume processing by a mainframe user job. An end-of-record RCW will be generated by TDEM1 to complete any partial sector of data residing in the Buffer Memory read-ahead area.

The end-of-data read request (FC$EODR) is issued to generate the end-of-file and end-of-data RCWs that terminate a user dataset or label data stream.

End read requests are passed by BCOM3 to the BYPASS activity for processing on the DATQU queue in the XIOP Kernel. BYPASS queues the request to the appropriate TEX activity for the requested device. BYPASS handles stacking of the user and label DSC tables by calling the DSCGET routine. TEX calls the TAPEND routine for the end read processing.

Figure 4-5 shows the processing of end read requests.

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If it is not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the BYPASS activity on the DATQU in the Kernel. BYPASS is activated, if waiting, by popping the TIMQU in the Kernel.

BYPASS - Dequeues the next request from DATQU and locates the TCB for the requested device. The Hold Data flag (DD@HLD) in the request is examined. If it is set and the user DSC table has not been saved, DSCGET is called. Likewise, if the Hold Data flag is not set and the user DSC table is being held (DC@DHU, DC@DHL), DSCGET is called to restore the stacked DSC table (DC@DSU, DC@DSL).

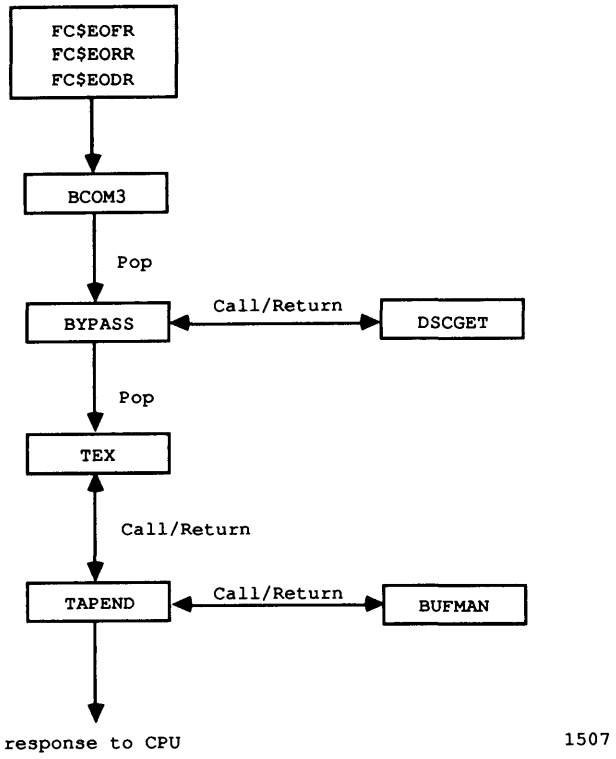


Figure 4-5. Processing of End Read Requests

BYPASS (continued) - An End Read request from the mainframe may include a request to append (DD@APP) the contents of a held data stream (DD@HLD) to the active data stream. If BYPASS finds no hold data stream (DC@DHU, DC@DHL), a protocol error response is returned to the mainframe.

BYPASS activates the TEX activity by popping DC@QUA and placing the mainframe request on the DC@MSG queue.

BYPASS processes all requests queued to it by BCOM3 and waits on TIMQU when finished.

TEX - Calls the TAPEND routine to initiate End Read processing for the request.

TAPEND - Begins processing by checking the Append Data flag (DD@APP) in the request. If it is set, the write-behind data described by the held DSC (DC@DHU, DC@DHL) is appended to the active read data stream (DC@DSU, DC@DSL).

TAPEND (continued) - Write-behind data is appended by copying the range of buffer descriptor entries described by the XIOP block pointer (CU@PTR) and the BIOP block pointer (NX@PTR) in the held DSC header to the active DSC at the bottom pointer (CU@BTM). A protocol error check is made to ensure that all descriptors copied from the held DSC will fit in the active read DSC. The count of data sectors in the read-ahead area (CU@VMS) is adjusted by the amount of data appended, plus any control words that will be added by TDEM1 when the data is moved to Central Memory.

After any append processing is complete, TAPEND checks the function code in the request for an end-of-file or end-of-data read. If either is requested, a call to BUFMAN is made to allocate a buffer descriptor entry for the EOF or EOD block to be added.

BUFMAN - Before allocating a buffer for the request, BUFMAN deallocates buffers for any data previously transferred to the mainframe. The top and BIOP (NX@PTR) pointers mark this range of descriptors. After deallocation, the top pointer is adjusted to equal the BIOP pointer.

BUFMAN allocates a single buffer, based on the function code in the request being other than read or write. A check is made to determine if the descriptor for the buffer will fit in the DSC. If not, an error code (1) is returned to TAPEND. If a single buffer is not available, an error code (2) is returned to TAPEND.

TAPEND - Returns a protocol error to the mainframe if an error status is returned from BUFMAN.

If allocation was successful, the descriptor for the buffer just allocated is marked with the appropriate status (BF@STA) for end-of-file (MD$EOF) or end-of-data (MD$EOD). The count of sectors in the read-ahead area (CU@VMS) is adjusted to include the control words to be added by TDEM1 when the data is transferred to Central Memory.

If an end-of-record read is requested instead of EOF or EOD, the last partial block, if any, in the read-ahead area is to be completed. A call to BUFMAN is not needed because the descriptors for the partial block are already present in the DSC. The status field (BF@STA) for each descriptor of the partial block is marked with the end-of-record (MD$EOV) status. This causes TDEM1 in the BIOP to set the Null flag in the end of record control word for the block when the partial sector containing the end of the block is transferred to Central Memory.

TAPEND (continued) - TAPEND generates a response containing the count of data sectors in the read-ahead area (TQ@VMS) and the number of unallocated buffers in Buffer Memory (TQ@MOS), and sends it to the mainframe before returning to TEX.

TEX - Waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.6 NO-OP REQUEST (FC$NOOP)

The No-op request is issued by the mainframe tape driver to guarantee that all outstanding mainframe requests have been processed by the tape subsystem. The request is typically used to clear any outstanding data transfer requests after an error or end-of-file condition is reported.

No-op requests are passed by BCOM3 to the BYPASS activity for processing on the DATQU queue in the XIOP Kernel. BYPASS queues the request to the appropriate TEX activity for the requested device. BYPASS handles stacking of the user and label DSC tables by calling the DSCGET routine. TEX handles discarding of data in the user and label DSC tables by calling TAPDIS.

Figure 4-6 shows the processing of a no-op request.

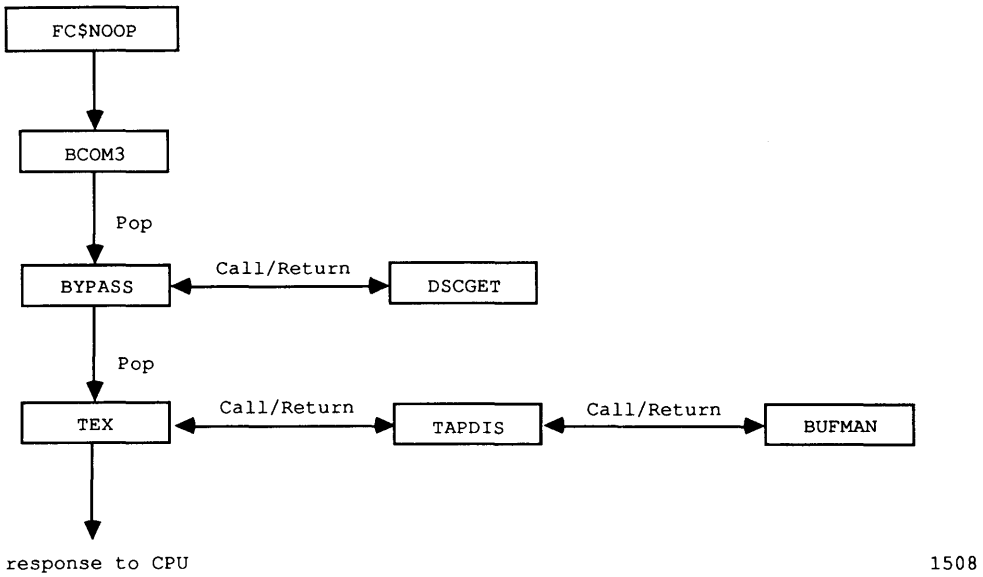


Figure 4-6. Processing of No-op Requests

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If no TCB is present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the BYPASS activity on the DATQU in the Kernel. BYPASS is activated, if waiting, by popping the TIMQU in the Kernel.

BYPASS - Dequeues the next request from DATQU and locates the TCB for the requested device. The Hold Data flag (DD@HLD) in the request is examined. If it is set and the user DSC table has not been saved, DSCGET is called. If the Hold Data flag is not set and the user DSC table is being held (DC@DHU, DC@DHL), DSCGET is called to restore the stacked DSC table (DC@DSU, DC@DSL).

The residual sector count (DC@RSC) in the TCB is cleared to prevent any new data transfer requests to TDEM1 by TAPEIO.

BYPASS activates the TEX activity by popping DC@QUA and placing the mainframe request on the DC@MSG queue.

BYPASS processes all requests queued to it by BCOM3 and waits on TIMQU when finished.

TEX - Calls the TAPDIS routine to deallocate buffers for the active or held DSCs if the Discard User Data flag (DD@DUD), or Discard Label Data flag (DD@DLD) is set in the request. TEX waits for all data transfers in progress to complete before calling TAPDIS by pushing on the DC@QUB queue in the TCB.

TAPDIS - Calls BUFMAN to discard the data buffers for the user data DSC and label DSC. The user data DSC may be the active DSC or the held DSC. TAPDIS determines where the user DSC can be found and passes the appropriate parameter (FN$DISC, FN$DISCH) to BUFMAN. The label DSC can only be present when a user DSC is being held.

BUFMAN - Deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - Generates a response with a 0 status field (TQ@STS) and sends it to the mainframe.

TEX waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.7 POSITIONING REQUESTS (FC$FWFIL, FC$FWSPC, FC$BKFIL, FC$BKSPC)

The mainframe driver may issue tape positioning requests to the tape subsystem to forward or backward space some number of files or blocks on the tape. Tape marks delimit file boundaries during positioning.

Positioning requests are passed by BCOM3 to the appropriate TEX activity for processing on the DC@MSG queue in the TCB for the requested device. BCOM3 pops the TEX activity waiting on the DC@QUA queue in the TCB. TEX handles discarding of data in the user and/or label DSC tables by calling TAPDIS. TEX calls TAPMOV to process the position request.

Figure 4-7 shows the processing of positioning requests.

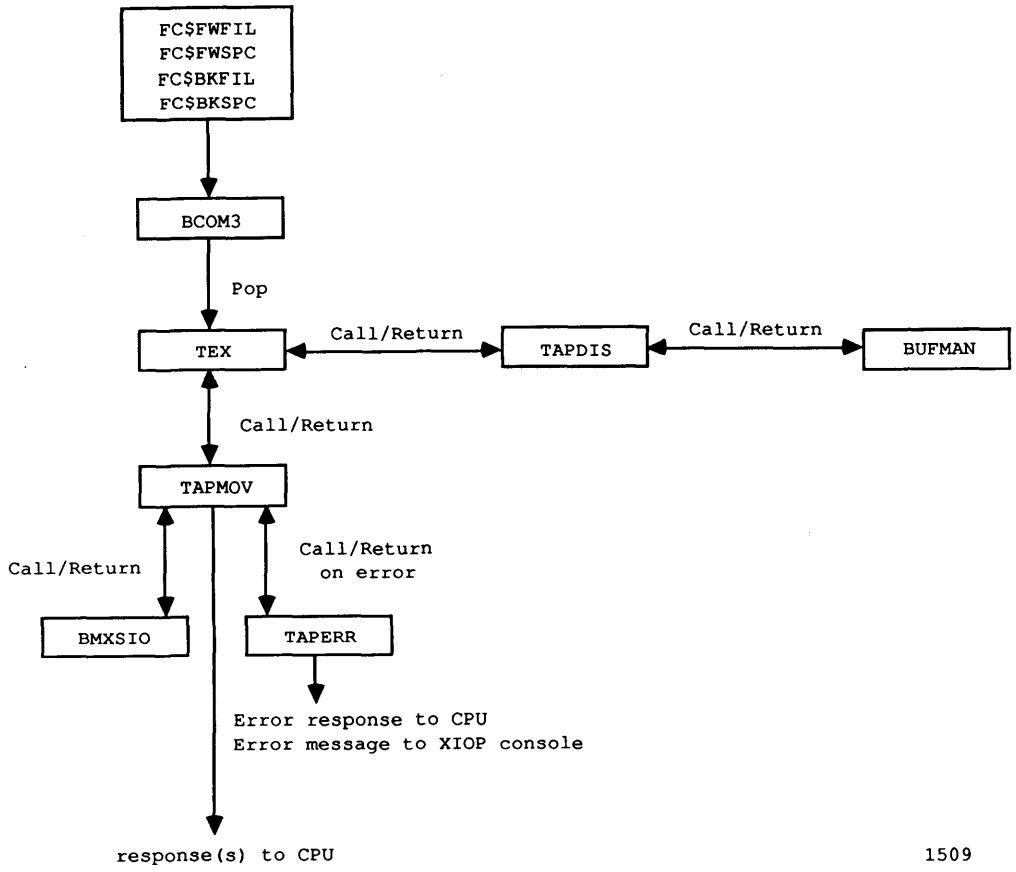


Figure 4-7. Processing of Positioning Requests

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the TEX activity on DC@MSG queue in the TCB. TEX is activated, if waiting, by popping the DC@QUA queue in the TCB.

TEX - If the Discard User Data flag (DD@DUD) or Discard Label Data flag (DD@DLD) is set in the request, TEX calls the TAPDIS routine to deallocate buffers for the active or held DSCs. TEX waits for all data transfers in progress to complete before calling TAPDIS by pushing on the DC@QUB queue in the TCB.

TAPDIS - Calls BUFMAN to discard the data buffers for the user data DSC and label DSC. The user data DSC may be the active DSC or the held DSC. TAPDIS determines where the user DSC can be found and passes the appropriate parameter (FN$DISC, FN$DISCH) to BUFMAN. The label DSC can only be present when a user DSC is being held.

BUFMAN - Deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - Calls TAPMOV to process the positioning request.

TAPMOV - Provides the nondata I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPMOV uses the requested block count (TQ@RBC) in the packet to determine the number of blocks or files to be skipped.

TAPMOV builds single positioning commands for the BMX subsystem. Each command is stored in a Channel Program Word (CPW@) structure in the CPB. TAPMOV calls BMXSIO once for each block or file to be skipped.

TAPMOV checks the operation status (OS$) when BMXSIO returns.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPMOV calls BMXSIO to restart the I/O.

TAPMOV (continued) - An operation status of OS$IP implies that the command is not complete. The control unit normally presents an initial status to the BMX subsystem on positioning commands that may take a long time to complete. The initial status indicates that the control unit has disconnected from the channel and will present ending status when the command is complete. This initial channel end status is reflected back to TAPMOV as an OS$IP operation status. TAPMOV calls BMXSIO to wait for the ending status of the command.

A status of OS$DN implies that the command completed successfully. TAPMOV checks for additional blocks or files to skip. If the requested count is not exhausted, TAPMOV calls BMXSIO to repeat the command. If the request is complete, TAPMOV generates a final response to the mainframe and returns to TEX.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active command in progress. TAPMOV generates a response to the mainframe with the not ready status bit (ST@RDY) set and returns to TEX.

Finally, an operation status of OS$ER indicates that the BMX subsystem detected an error on the last position command. The error must be related to a hardware condition (channel error, unit check, unit exception) because no data I/O is active.

TAPMOV detects the unit exception condition encountered by examining the ending device status (CPB@DS) for the unit exception status bit (ST$UE). An end-of-file status (ST@EOF) is returned in the mainframe response if a space block function was requested. TAPMOV returns to TEX in this case.

The unit exception condition is normal status for file positioning functions. Processing continues if additional files are to be skipped.

Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the ending device status (CPB@DS) has the unit check status bit (ST$UC) set. If either type of error is present, TAPMOV calls TAPERR to create an error recovery activity to retry the failed command.

TAPERR returns the status of the recovery attempt. A zero status indicates the command was recovered successfully.

TAPMOV (continued) - Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPMOV returns to TEX.

TEX - If TAPMOV detected an error, TEX sets the error flag (DC@ERR) to invoke the next valid packet mechanism.

TEX waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.8 LOAD DISPLAY REQUEST (FC$DSP)

The mainframe driver issues the load display request to the tape subsystem to display a volume serial number (VSN) on the display panel of a cartridge type tape device. The load display request can also be used to clear a previous VSN shown on the display.

Load display requests are passed by BCOM3 to the appropriate TEX activity for processing on the DC@MSG queue in the TCB for the requested device. BCOM3 pops the TEX activity waiting on the DC@QUA queue in the TCB. TEX handles discarding of data in the user and label DSC tables by calling TAPDIS. TAPMOV is called to process the display request.

Figure 4-8 shows the processing of a display request.

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the TEX activity on DC@MSG queue in the TCB. TEX is activated, if waiting, by popping the DC@QUA queue in the TCB.

TEX - If the Discard User Data flag (DD@DUD) or Discard Label Data flag (DD@DLD) is set in the request, TEX calls the TAPDIS routine to deallocate buffers for the active or held DSCs. TEX waits for all data transfers in progress to complete before calling TAPDIS by pushing on the DC@QUB queue in the TCB.

TAPDIS - Calls BUFMAN to discard the data buffers for the user data DSC and label DSC. The user data DSC may be the active DSC or the held DSC. TAPDIS determines where the user DSC can be found and passes the appropriate parameter (FN$DISC, FN$DISCH) to BUFMAN. The label DSC can only be present when a user DSC is being held.

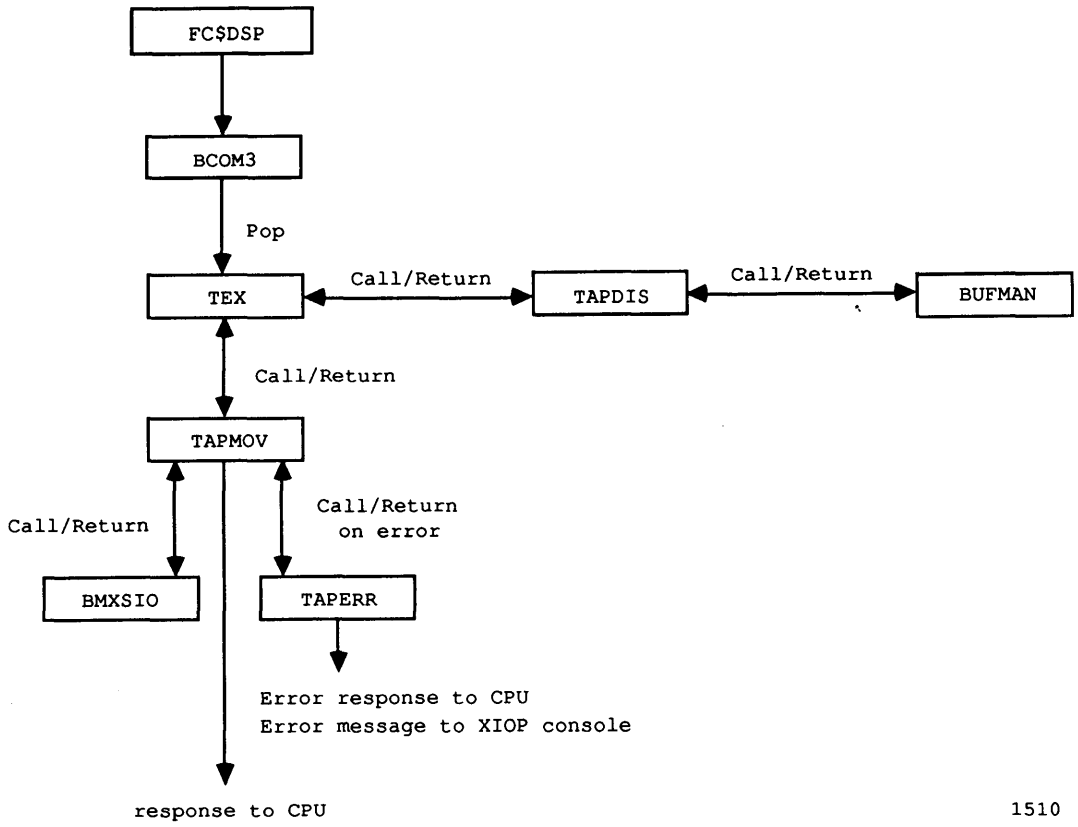


Figure 4-8. Processing of Display Requests

BUFMAN - Deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - Ensures that the type of the requested device is a cartridge tape unit. If not, a protocol error response is returned to the mainframe.

TEX calls TAPMOV to process the display request.

TAPMOV - Provides the nondata I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPMOV (continued) - TAPMOV copies the display data from the request packet to a Local Memory scratch buffer. Up to 16 bytes of data, preceded by a function code byte, may be displayed.

TAPMOV builds a single load display command for the BMX subsystem. The command is stored in a Channel Program Word (CPW@) structure in the CPB. TAPMOV calls BMXSIO to issue the device command.

TAPMOV checks the operation status (OS$) when BMXSIO returns.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPMOV calls BMXSIO to restart the I/O.

An operation status of OS$IP or OS$BZ implies that the command is not complete. TAPMOV calls BMXSIO to wait for the ending status of the command.

A status of OS$DN implies that the display command completed successfully. TAPMOV sends a response packet to the mainframe and returns to TEX.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active command in progress. TAPMOV generates a response to the mainframe with the not ready status bit (ST@RDY) set and returns to TEX.

Finally, an operation status of OS$ER indicates that the BMX subsystem detected an error on the load display command. The error must be related to a hardware condition (channel error, unit check, unit exception) because no data I/O is active.

Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the ending device status (CPB@DS) has the unit check status bit (ST$UC) set. If either type of error is present, TAPMOV calls TAPERR to create an error recovery activity to retry the failed command.

TAPERR returns the status of the recovery attempt. A status of 0 indicates that the command was recovered successfully. Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPMOV returns to TEX.

TEX - Waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.9 REMOUNT REQUEST (FC$RMNT)

The mainframe driver issues the remount request to the tape subsystem when the end-of-volume, in a multiple volume dataset, is encountered. A remount request is also issued when an unrecovered write error forces premature termination of a tape volume. The remount request differs from the mount request in that the data streams associated with the original device are carried over to the new device.

Remount requests are passed by BCOM3 to the appropriate TEX activity for processing on the DC@MSG queue in the TCB for the requested device. BCOM3 pops the TEX activity waiting on the DC@QUA queue in the TCB. TEX calls the BMXOPE routine in the BMX subsystem to process the request. BMXOPE calls BMXTPO for the actual mount processing.

Figure 4-9 shows the processing of a remount request to the same device. Figure 4-10 shows the processing of a remount request to a new device.

BCOM3 - BCOM3 checks the Device Table Open flag (BDV@OP) to see if a TEX activity exists for the requested device. If not open, the BMXOPE activity is created to process the request. If the activity cannot be created, a protocol error response is sent to the mainframe.

If the requested device is open, BCOM3 checks for a TCB present (BDV@CP). If not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the TEX activity on DC@MSG queue in the TCB. TEX is activated, if waiting, by popping the DC@QUA queue in the TCB.

TEX - TEX calls the BMXOPE routine to process the request to remount to the original device.

BMXOPE - BMXOPE examines the previous device ordinal (TQ@PDV) in the request to determine if a remount is requested on the original device or a new device.

If a new device is requested, BMXOPE checks to see if a device activity currently owns the device (BDV@AI). This can occur if a configuration change is taking place on the device. If a device activity exists, BMXOPE waits for it to release the device, and assigns itself as the new device activity.

BMXOPE marks the device open (BDV@OP) and does a Goto to the BMXTPO routine for mount processing.

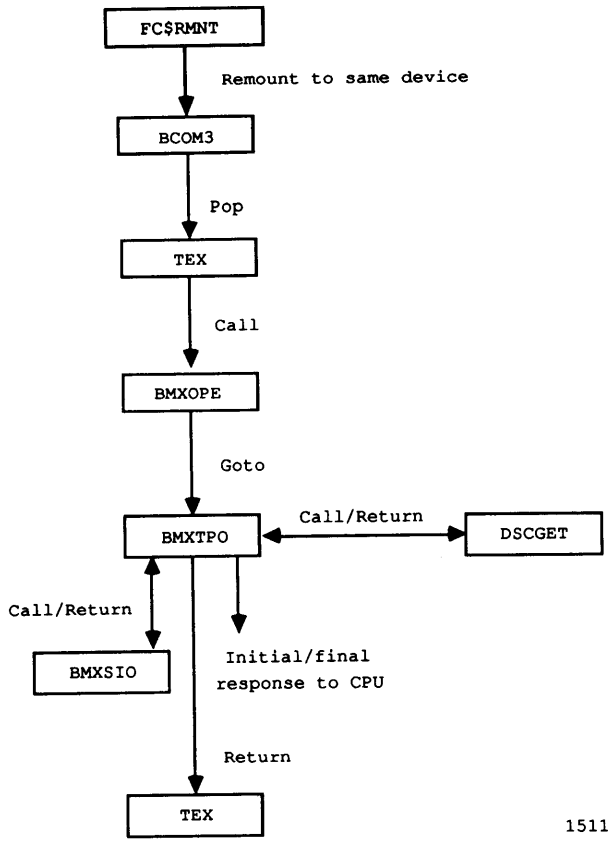


Figure 4-9. Processing of a Remount Request to the Same Device

BMXTPO - If a remount to a new device is requested, BMXTPO allocates control tables for the TEX activity. A TCB table is allocated (DC@) which contains the Command Parameter Block (CPB@) used to interface to the BMX subsystem. If Data Stream Control tables (DSCs) are present for the original device, their addresses are moved to the new TCB. Otherwise, a new DSC table is allocated by a call to DSCGET.

If a remount to the original device is requested, the TCB associated with the original device is reused. A new DSC is allocated by a call to DSCGET, if one is not already present for the device.

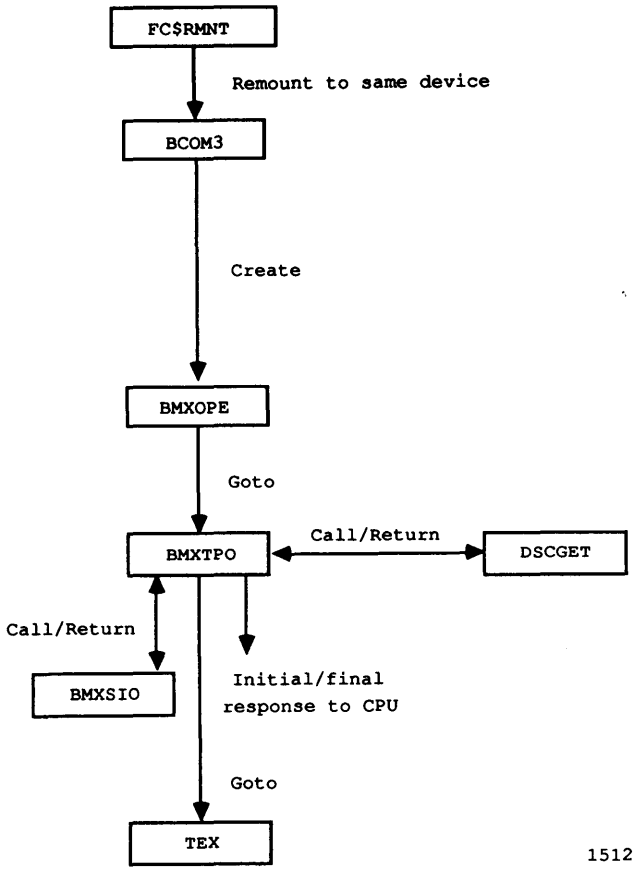


Figure 4-10. Processing of a Remount Request to a New Device

BMXTPO (continued) - BMXTPO arms the drive for load point. If the drive is not ready with a mounted tape at load point, an initial response indicating the not ready status (ST@RDY) is returned to the mainframe. When the drive is ready with a tape at load point, final status (ST@BOT) is sent to the mainframe along with write protect status (ST@NRW).

If the remount is to the original device, BMXTPO returns to the original TEX activity. Otherwise, BMXTPO does a Goto TEX to become the activity for the new device.

TEX - Waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.10 REWIND REQUESTS (FC$REWND, FC$RWND1, FC$RWND2)

The mainframe driver issues rewind requests to the tape subsystem to cause a mounted tape to be rewound to load point. The FC$RWND1 and FC$RWND2 functions specify that one or two tapemarks should be written before the rewind.

Rewind requests are passed by BCOM3 to the appropriate TEX activity for processing on the DC@MSG queue in the TCB for the requested device. BCOM3 pops the TEX activity waiting on the DC@QUA queue in the TCB. TEX handles discarding of data in the user and label DSC tables by calling TAPDIS. TAPMOV is called to process the position request.

Figure 4-11 shows the processing of rewind requests.

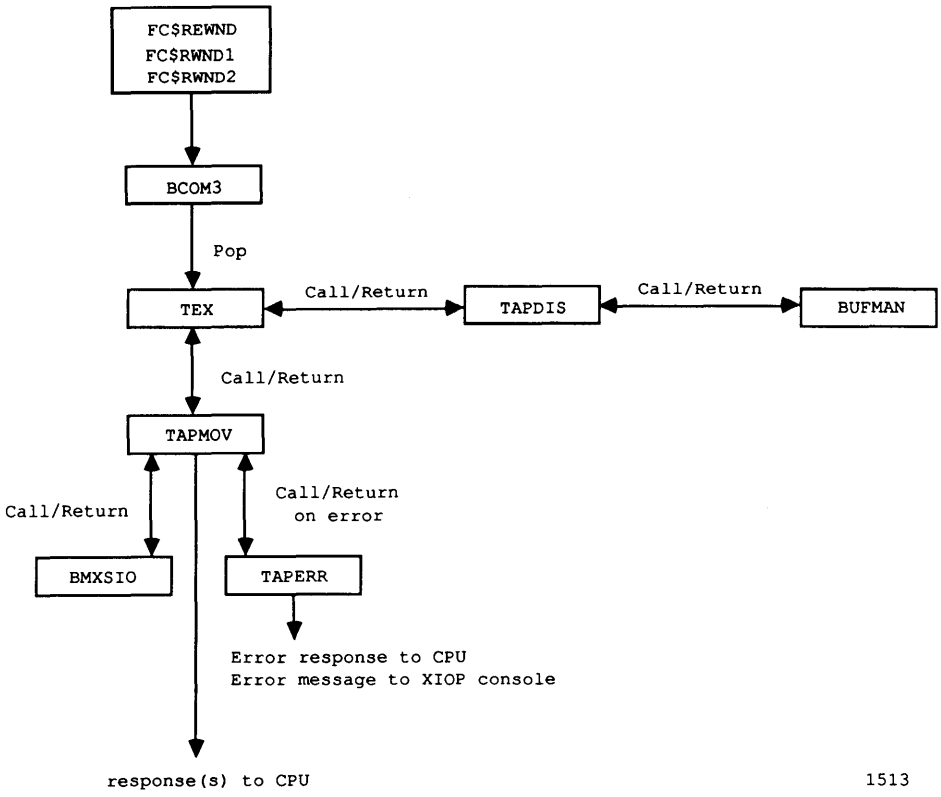


Figure 4-11. Processing of Rewind Requests

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If it is not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the TEX activity on DC@MSG queue in the TCB. TEX is activated, if waiting, by popping the DC@QUA queue in the TCB.

TEX - If the Discard User Data flag (DD@DUD) or Discard Label Data flag (DD@DLD) is set in the request, TEX calls the TAPDIS routine to deallocate buffers for the active or held DSCs. TEX waits for all data transfers in progress to complete before calling TAPDIS by pushing on the DC@QUB queue in the TCB.

TAPDIS - TAPDIS calls BUFMAN to discard the data buffers for the user data DSC and label DSC. The user data DSC may be the active DSC or the held DSC. TAPDIS determines where the user DSC can be found and passes the appropriate parameter (FN$DISC, FN$DISCH) to BUFMAN. The label DSC can only be present when a user DSC is being held.

BUFMAN - BUFMAN deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - TEX calls TAPMOV to process the rewind request.

TAPMOV - TAPMOV provides the nondata I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPMOV writes zero, one, or two tape marks before rewinding, based on the function requested.

TAPMOV builds write tape mark and rewind commands for the BMX subsystem. Each command is stored in a Channel Program Word (CPW@) structure in the CPB. TAPMOV calls BMXSIO once for each tape mark to be written and once for the rewind command.

TAPMOV checks the operation status (OS$) when BMXSIO returns.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPMOV calls BMXSIO to restart the I/O.

TAPMOV (continued) - An operation status of OS$IP implies that the command is not complete. The control unit normally presents an initial status to the BMX subsystem on rewind commands that may take a long time to complete. The initial status indicates that the control unit has disconnected from the channel and will present ending status when the command is complete. This initial channel end status is reflected back to TAPMOV as an OS$IP operation status.

TAPMOV generates an initial response packet to the mainframe with the Beginning-of-tape flag (ST@BOT) cleared. TAPMOV calls BMXSIO to wait for the ending status of the command.

A status of OS$DN implies that the device command completed successfully. For rewind commands, OS$DN status may instead indicate that the tape drive was manually reset and later made ready again.

TAPMOV returns a response to the mainframe containing the block finished status bit (ST@BTR) and EOF status bit (ST@EOF) for each tape mark written.

On completion of a rewind command, TAPMOV reads the sense bytes from the device to check that the tape is really at load point (SB$LPT). If the tape has not reached load point, TAPERR is called to determine and report the cause of the failure. If the rewind completes successfully, the beginning of tape status bit (ST@BOT) is returned to the mainframe. Otherwise, the error status from TAPERR is returned. TAPMOV returns to TEX.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active command in progress. TAPMOV generates a response to the mainframe with the not ready status bit (ST@RDY) set and returns to TEX.

Finally, an operation status of OS$ER indicates that the BMX subsystem detected an error on a write tape mark or rewind command. The error must be related to a hardware condition (channel error, unit check, unit exception), because no data I/O is active.

Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the ending device status (CPB@DS) has the unit check status bit (ST$UC) set. If either type of error is present, TAPMOV calls TAPERR to create an error recovery activity to retry the failed command.

TAPMOV (continued) - TAPERR returns the status of the recovery attempt. A status of 0 indicates that the command was recovered successfully.

Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPMOV returns to TEX.

TEX - If TAPMOV detected an error, TEX sets the error flag (DC@ERR) to invoke the next valid packet mechanism. TEX waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.11 UNLOAD REQUESTS (FC$UNLD, FC$UNLD1, FC$UNLD2)

The mainframe driver issues unload requests to the tape subsystem to cause a mounted tape to be rewound and unloaded. The FC$UNLD1 and FC$UNLD2 functions specify that one or two tapemarks should be written before the rewind and unload.

Unload requests are passed by BCOM3 to the appropriate TEX activity for processing on the DC@MSG queue in the TCB for the requested device. BCOM3 pops the TEX activity waiting on the DC@QUA queue in the TCB. TEX handles discarding of data in the user and label DSC tables by calling TAPDIS. TAPMOV is called to process the unload request.

Figure 4-12 shows the processing of unload requests.

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If it is not present, a protocol error response is sent to the mainframe. BCOM3 queues the request to the TEX activity on DC@MSG queue in the TCB. TEX is activated, if waiting, by popping the DC@QUA queue in the TCB.

TEX - If the Discard User Data flag (DD@DUD) or Discard Label Data flag (DD@DLD) is set in the request, TEX calls the TAPDIS routine to deallocate buffers for the active or held DSCs. TEX waits for all data transfers in progress to complete before calling TAPDIS by pushing on the DC@QUB queue in the TCB.

TAPDIS - Calls BUFMAN to discard the data buffers for the user data DSC and label DSC. The user data DSC may be the active DSC or the held DSC. TAPDIS determines where the user DSC can be found and passes the appropriate parameter (FN$DISC, FN$DISCH) to BUFMAN. The label DSC can only be present when a user DSC is being held.

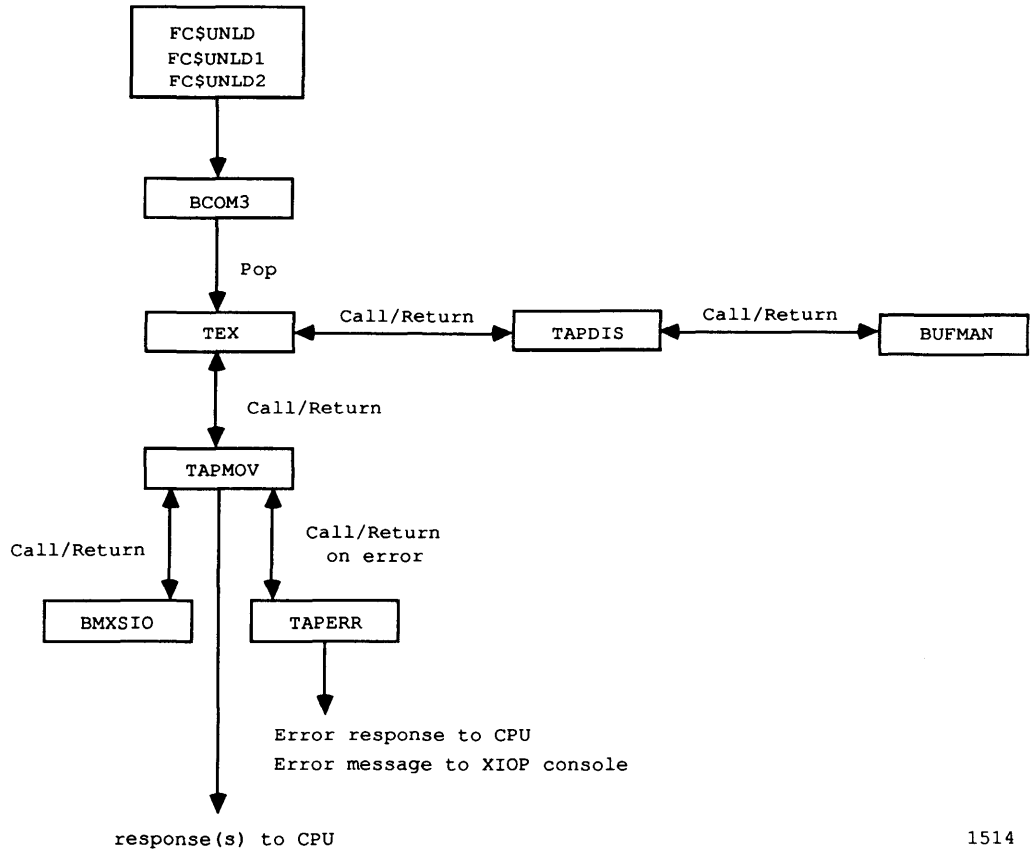


Figure 4-12. Processing of Unload Requests

BUFMAN - Deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - Calls TAPMOV to process the unload request.

TAPMOV - Provides the nondata I/O interface to the driver software in the BMX subsystem for the TEX activity. The interface uses the Command Parameter Block (CPB@) in the TCB and calls to the BMXSIO routine. The CPB contains the device command and response parameters that describe the I/O state during device processing.

TAPMOV writes zero, one, or two tape marks before unloading, based on the function requested.

TAPMOV (continued) - TAPMOV builds write tape mark and unload commands for the BMX subsystem. Each command is stored in a Channel Program Word (CPW@) structure in the CPB. TAPMOV calls BMXSIO once for each tape mark to be written and once for the unload command.

TAPMOV checks the operation status (OS$) when BMXSIO returns.

An operation status of OS$RT indicates that the device command should be retried. This typically occurs when Channel Command Retry status is detected by the BMX subsystem. TAPMOV calls BMXSIO to restart the I/O.

An operation status of OS$IP implies that the command is not complete. The control unit normally presents an initial status to the BMX subsystem on unload commands that may take a long time to complete. The initial status indicates that the control unit has disconnected from the channel and will present ending status when the command is complete. This initial channel end status is reflected back to TAPMOV as an OS$IP operation status.

TAPMOV generates an initial response packet to the mainframe with the Not Ready status flag (ST@RDY) cleared. TAPMOV calls BMXSIO to wait for the ending status of the command.

A status of OS$DN implies that the command completed successfully. For unload commands, OS$DN status may instead indicate that the tape drive was manually reset and later made ready again.

TAPMOV returns a response to the mainframe containing the block finished status bit (ST@BTR) and EOF status bit (ST@EOF) for each tape mark written.

OS$HD status indicates that the mainframe has issued an FC$FREE request to halt all processing on the device. BMXSIO will detect this condition and terminate any active command in progress. TAPMOV generates a response to the mainframe with the Not Ready status flag (ST@RDY) set and returns to TEX.

Finally, an operation status of OS$ER is reported by the BMX subsystem when an error occurs on a write tape mark command or as normal ending status for an unload command. (A unit check condition is reported by the device at completion of an unload.)

TAPMOV (continued) - Channel errors and software errors are detected when the CPB@EC field is nonzero. A unit check error is present if the ending device status (CPB@DS) has the unit check status bit (ST$UC) set. If either type of error is present for a write tape mark command, TAPMOV calls TAPERR to create an error recovery activity to retry the failed command.

The interpretation of error status for an unload command is a bit more complicated. TAPERR is called to handle any channel errors or software errors. A unit check error on initial status from the device is considered an error condition. TAPERR is called to determine and report the cause of the error. If a unit check error is presented by the device on ending status, TAPMOV reads the sense bytes for the device to check the Intervention Required sense bit (SB$IVR). If the bit is set, the unload completed successfully. The Not Ready status flag (ST@RDY) is set in the response packet and sent to the mainframe to indicate the unload request is complete. If the Intervention Required sense bit is not set, the unload may not have completed successfully. In this case, TAPERR is called to determine the cause and report the error.

TAPERR returns the status of the recovery attempt. A zero status indicates the command was recovered successfully.

Any other status returned by TAPERR is considered unrecovered and is returned to the mainframe in the response packet (TQ@STS). TAPMOV returns to TEX.

TEX - TEX sets the error flag (DC@ERR) to invoke the next valid packet mechanism if TAPMOV detected an error.

TEX waits for the next mainframe request (DC@MSG) by pushing on the request queue (DC@QUA) in the TCB.

4.3.12 FREE REQUEST (FC$FREE)

The Free request is issued by the mainframe tape driver to terminate processing and close a device. All resources allocated at mount time are released at close.

Free requests are passed by BCOM3 to the BYPASS activity for processing on the DATQU queue in the XIOP Kernel. BYPASS queues the request to the appropriate TEX activity for the requested device. BYPASS handles stacking of the user and label DSC tables by calling the DSCGET routine. TEX handles discarding of data in the user and/or label DSC tables by calling TAPDIS. TEX calls BMXOPE for close processing.

Figure 4-13 shows the processing of free requests.

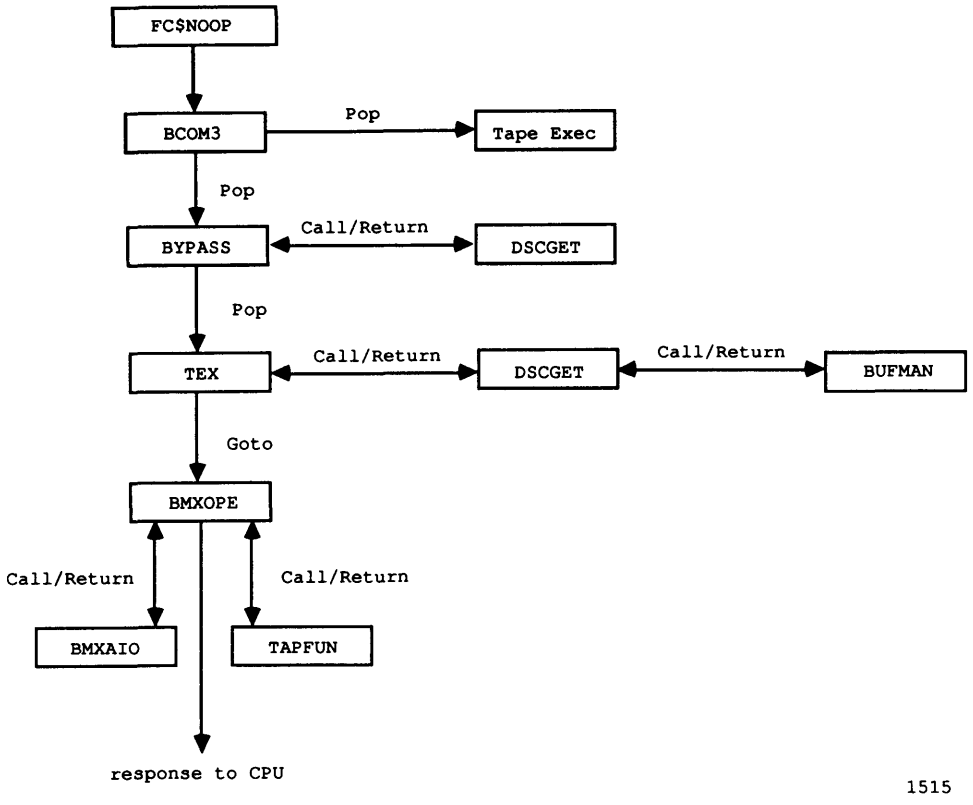


Figure 4-13. Processing of Free Requests

BCOM3 - Checks for a TCB present for the requested device (BDV@CP). If it is not present, a protocol error response is sent to the mainframe.

Some routine of the Tape Exec activity for the device may be waiting for a prior I/O or mount request to complete. In this case, the Tape Exec activity will be pushed on the task wait queue (BDV@TQ) of the device table. BCOM3 pops the Tape Exec activity, if waiting. The Free Pending flag (CPB@FP) is set in the CPB section of the TCB to signal that processing should terminate.

BCOM3 queues the request to the BYPASS activity on the DATQU in the Kernel. BYPASS is activated, if waiting, by popping the TIMQU in the Kernel.

BYPASS - Dequeues the next request from DATQU and locates the TCB for the requested device. The Hold Data flag (DD@HLD) in the request is examined. If the flag is set and the user DSC table has not been saved, DSCGET is called. If the Hold Data flag is not set and the user DSC table is being held (DC@DHU, DC@DHL), DSCGET is called to restore the stacked DSC table (DC@DSU, DC@DSL).

The residual sector count (DC@RSC) in the TCB is cleared to prevent any new data transfer requests to TDEM1 by TAPEIO.

BYPASS activates the TEX activity by popping DC@QUA and placing the mainframe request on the DC@MSG queue.

BYPASS processes all requests queued to it by BCOM3 and waits on TIMQU when finished.

TEX - Calls DSCGET to deallocate any active or held DSC present in the TCB. TEX waits for all data transfers in progress to complete before calling DSCGET by pushing on the DC@QUB queue in the TCB.

DSCGET - Calls BUFMAN to release all buffers for the DSC (DC@DSU, DC@DSL). DSCGET then releases the Buffer Memory for the DSC.

BUFMAN - Deallocates all buffers described by the range of descriptors between the top (CU@TOP) and bottom (CU@BTM) pointers in the XIOP DSC header. After deallocation, the top, XIOP (CU@PTR), and BIOP (NX@PTR) pointers are adjusted to equal the bottom pointer. The remainder of the XIOP and BIOP sections of the DSC header are cleared.

TEX - Clears the Free Pending flag (CPB@FP) in the CPB section of the TCB. TEX does a Goto BMXOPE to close the device.

BMXOPE - Reads the sense bytes for the device to relieve any contingent connection caused by an earlier unit check. The cartridge display panel is cleared, if appropriate. A selective reset command is issued for the device by a call to BMXAIO. BMXOPE releases the Local Memory for the TCB associated with the device, and sends a response to the mainframe for the free request.

The Tape Exec activity terminates.

4.4 ERROR RECOVERY PROCESSING

Each device activity initiates error recovery by calling the TAPERR overlay. Error recovery may also be called recursively when new errors are encountered during recovery. Alternating or cyclic errors could cause an indefinite number of error recovery activities to be created. Kernel SMOD storage for activities would quickly be exhausted in such a situation. The RCV$MAX parameter in APTEXT is used to limit the number of recursive calls during error recovery. TAPERR returns the nonoperational status (ST@NOP) when this limit is reached. The count of error recovery levels in progress is stored in the TCB (DC@LEV). The calling device activity and each subsequent level of error recovery wait for a response on the DC@QUC queue in the TCB. Activities are queued in LIFO order.

4.4.1 TAPERR ROUTINE

The TAPERR routine handles creation of error recovery activities. It enforces the maximum level of recovery allowed. TAPERR returns the ending recovery status to the calling routine. The TERROR or TCART routine is created as the first overlay of each error recovery activity based on the type of device in error. TCART is used for cartridge devices and TERROR is used for noncartridge devices.

4.4.2 TERROR ROUTINE

The TERROR routine controls noncartridge device recovery. It is responsible for decoding the type of error encountered, calling the appropriate error subroutine, calling the TRTELL routine to display an error message on the XIOP console, and creating an error packet to be sent to the mainframe system log file.

TERROR begins by checking for a channel error or software error (CPB@EC). A software overrun error may have occurred on a device command that is still active. TERROR calls BMXAIO to halt any device command that may be in progress.

The ending device status is then examined for a unit check error (ST$UC). If a unit check error is present, it may or may not be accompanied by the device end status (ST$DE). The lack of device end status occurs when a device has gone not ready, typically during a positioning or control command. The absence of the device end status indicates that the device will present ending status when readied. TERROR signals the BMX subsystem to throw away the device ready status by clearing the Request-in Expected flag (BDV@RI) in the device table. This prevents the BMX subsystem from reassigning ownership of a control unit path to the device, because the device activity may terminate before the unit is readied.

If not presented by the calling routine, the sense bytes for the device are read. Each sense bit is checked in priority order by TERROR. The sense bits are listed below in priority order along with the routine that handles recovery for the error.

Sense Bit	Routine
Equipment check	TREQC
Bus out check	TRBOC
Intervention required	TRINR
Command reject	TERROR
Hardware overrun	TRORN
Load point detected	TERROR
Data check	TRDCK
Data security erase	TERROR
Data converter check	TERROR
Not capable	TERROR
Id burst check	TRIDB

If none of the device detected errors above are present, the TRCER routine is called for recovery of timeouts, software, or channel errors.

4.4.3 TCART ROUTINE

The TCART routine controls cartridge device recovery. It is responsible for decoding the type of error encountered, retrying the device command in error, calling the TCTELL routine to display an error message on the XIOP console, and creating multiple error packets to be sent to the mainframe system log file. Multiple error packets are needed because the sense bytes for cartridge devices will not fit in a single packet.

TCART begins by checking for a channel error or software error (CPB@EC). A software overrun error may have occurred on a device command that is still active. TCART calls BMXAIO to halt any device command that may be in progress.

If not presented by the calling routine, the sense bytes for the device are read. Sense byte 3 specifies the error recovery procedure (ERP) to be used. Based on the ERP code and command, retries may be attempted by calling the error recovery I/O routines (TRWRT, TRRDF, TRRDB, or TAPFUN).

If no device-detected errors are specified by the ERP code, the TRCER routine is called for recovery of timeouts, software, or channel errors.

4.4.4 RECOVERY SUBROUTINES

The following subsections describe the subroutines called by TERROR or TCART to perform specific error recovery tasks.

4.4.4.1 Equipment check (noncartridge device only)

Equipment checks are unrecoverable. They generally point to a mechanical problem that prevents tape motion.

TERROR calls TREQC. If Device End is not set, control returns immediately with a status telling TERROR to ignore the equipment check and go on to the next sense bit. If device end is set, TREQC checks the sense bytes to see if reset was hit or if tape indicate is set. A reset hit is reported by the ST@RST status. Tape indicate is reported as a tape off the end of its reel (ST@LST). If neither is set, nonoperational status (ST@NOP) is returned.

4.4.4.2 Bus-out check (noncartridge device only)

Bus out checks refer to parity errors between the channel and the control unit. If device end is set and the command is a write, the tape is backspaced to the start of record (TAPFUN). Recovery is attempted by retrying the command six times before returning a nonoperational (ST@NOP) status. Retry is accomplished by calling the subroutine responsible for the failing command as follows:

Command	Subroutine
Read forward	TRRDF
Read reverse	TRRDB
Write	TRWRT
Others	TAPFUN

4.4.4.3 Intervention required (noncartridge device only)

Intervention required means manual intervention is required to correct the condition.

TERROR calls TRINR for processing. TRINR checks for device end; if set, the return status tells TERROR to keep checking sense bytes. If the current function is a rewind (CM$RWD) or unload (CM$RWU), intervention required is ignored. Otherwise, TRINR checks to see if reset was hit or if the tape went off the end of the reel, and it sets the appropriate status.

4.4.4.4 Command reject, data converter check, and not capable

Command reject, data converter check, and not capable errors are all unrecoverable. TERROR or TCART determines the appropriate status to return. Command reject errors are checked for the presence of a write ring during a write. If the write ring is missing, a ST@NRW status is returned; otherwise, nonoperational (ST@NOP) is returned. Data converter checks are returned as nonoperational (ST@NOP). Not capable is returned as ST@NCP.

4.4.4.5 Data overrun (noncartridge device only)

Data overrun occurs when the channel cannot keep up with data flow to or from the control unit. Recovery consists of retrying the I/O six times before returning unrecoverable error status. Retries are accomplished by repositioning the tape (TAPFUN) and calling the appropriate subroutine based on the command that failed.

4.4.4.6 Load point

Load point is set when a load point marker is sensed on the tape. TERROR or TCART returns load point status (ST@BOT), unless load point was encountered during an error recovery command.

4.4.4.7 Data check

Data check is set when an error is detected on the data being written to or read from the tape. Recovery depends on the command in effect.

Write (noncartridge device only) - Write recovery consists of repositioning the tape (TAPFUN), issuing an erase (TAPFUN), and retrying the write (TRWRT). Fifteen attempts are made before returning an unrecovered error.

Read (noncartridge device only) - Read recovery consists of repositioning the tape (TAPFUN) and retrying the read (TRRDF). The read is retried 41 times with a tape cleaner sequence (TRCLN) issued after every fourth retry. The tape cleaner sequence consists of moving the tape back over the tape cleaner and then repositioning for the next retry. If the error persists, read reverse recovery (TRRDB) is given 41 retries before unrecovered error status is returned.

4.4.4.8 Data security erase

TERROR or TCART returns unrecoverable status (ST@NOP).

4.4.4.9 ID burst check (noncartridge device only)

ID burst check is set when an error occurs while writing the ID burst off load point. Recovery consists of issuing a rewind (TAPFUN) and retrying the command. Write commands are retried 15 times. Write tapemark and erase commands are retried 16 times.

4.4.5 ERROR DISPLAY

After the error recovery subroutine returns to TERROR or TCART, TERROR or TCART calls the TRTELL or TCTELL overlay to record the error recovery information in Buffer Memory and display a message on the XIOP Kernel console. The message is in the following format:

hh:mm:ss *err* *cmd* *rtc* *chn* *dev* *sta*

hh:mm:ss Time when error occurred

err Type of error. See the list of tape device error messages in the IOS operator's guides.

cmd Command in effect when error was detected. (See the list of commands for tape device error messages in the IOS operator's guides.)

rtc Number of retries issued to recover the error (decimal)

chn Channel on which the error was detected (octal)

dev Control-unit/device address at which the error was detected

sta Ending status:
RECOVERED
UNRECOVERED

5. BLOCK MULTIPLEXER CHANNEL INTERFACE

5.1 IOS BLOCK MUX (BMX) SUBSYSTEM OVERVIEW

The IOS Block Mux (BMX) subsystem is composed of the following software routines, which are necessary to support the connection of IBM-compatible devices to Cray computer systems:

Routine	Description
BMXSIO	The BMXSIO overlay is the device driver. It is the interface between device activities and the BMX channel driver (see figure 5-1). It is responsible for: • Assigning the device path • Initiating I/O for the device • Returning status to the device activity
BMXDEM	The BMXDEM overlay is a demon activity responsible for all I/O between the channel and the device; see figure 5-1.
IBMX	The IBMX routine is the Kernel-resident interrupt handler for all BMX channel interrupts; see figure 5-1.
BMXCPU	The BMXCPU overlay is responsible for the BMX table structure and device configuration (see figures 5-2 and 5-3). These configurations are based on information in the Configuration Table (CNT) passed to the IOS by the CPU at startup time. See the COS Table Descriptions Internal Reference Manual, publication SM-0045, for a description of CNT.
BMXCON	The BMXCON overlay is responsible for configuring individual BMX device components up or down. The components include: • BMX channels • Attached control units • Attached devices
BMXOPE	The BMXOPE overlay is responsible for opening and closing devices attached to the BMX subsystem.

BMXTPO — The BMXTPO overlay is responsible for completing the open (tape mount) function.

These routines are described in more detail later in this section.

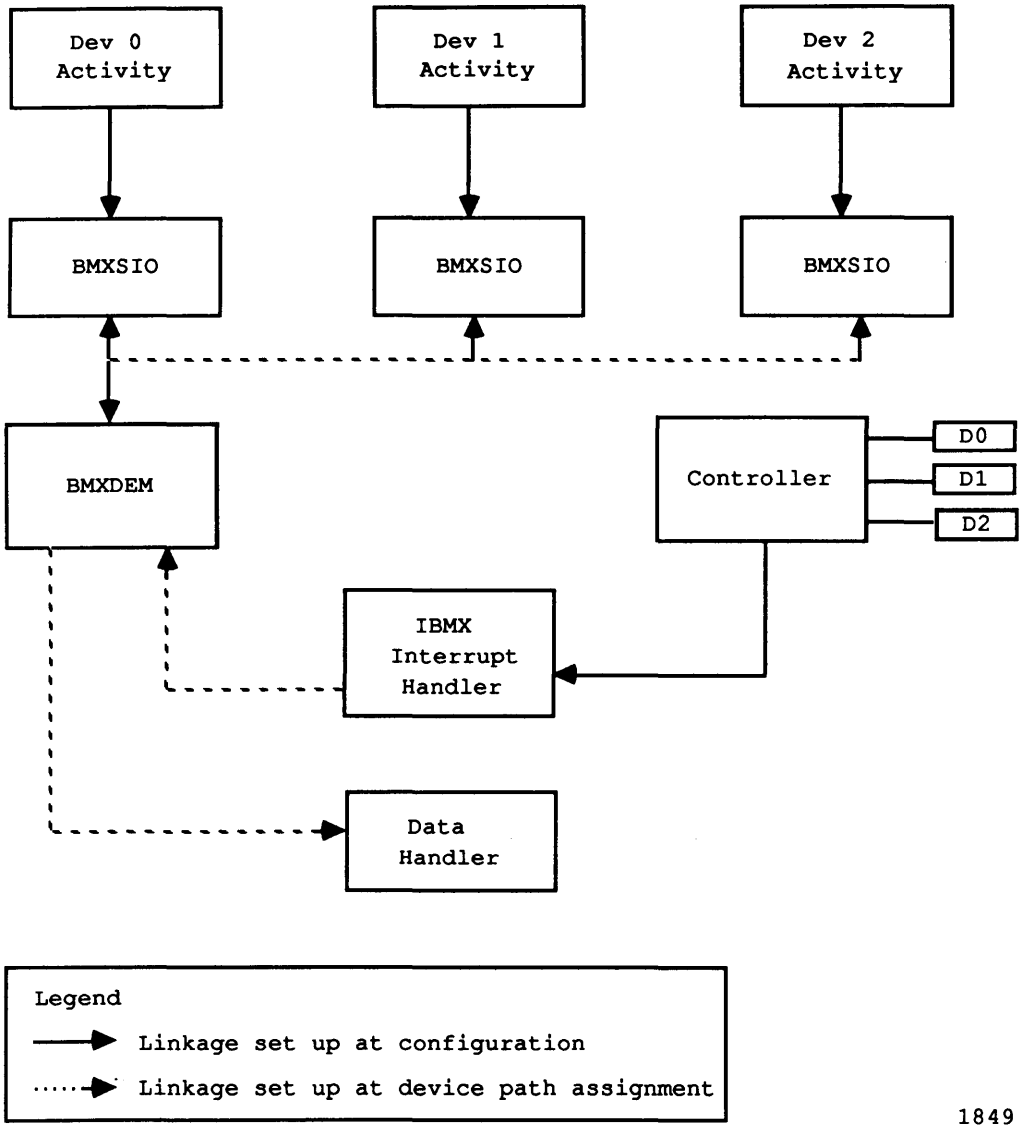


Figure 5-1. BMX Overview

5.2 BMX CONFIGURATION

The table structure that describes the BMX subsystem reflects the hardware configuration. There is a table for each component attached to the BMX subsystem, and, as shown in the following figures, the way the tables are structured corresponds to the interlinkages between the hardware components.

Three basic structures can be configured in a BMX subsystem, as follows:

- Multiple paths to a single bank
- Single path to multiple banks
- Multiple paths to multiple banks

A *path* is the channel/control-unit pair used to issue I/O to a device.

A *bank* is the set of control units all having access to the same set of devices. A bank with four control units that all access the same eight devices is represented as a 4-by-8 bank or configuration (4 control units, 8 devices).

The total BMX configuration may be a combination of one or more multiple-path, single-bank structures, along with one or more single-path, multiple-bank structures, or multiple-path, multiple-bank structures.

In figures 5-2, 5-3, and 5-4, the table structure diagrams represent the three basic structures that can be configured. In each figure, the hardware configuration is shown along with the table structure representing it. The solid lines in the table structure diagrams show the linkage that would be set up at configuration time; the broken lines show linkage set up at the time of path assignment to a device.

5.3 BMX TABLES

All BMX tables are allocated and set up by BMXCPU based on information in the CNT (Configuration Table) that is received from COS at startup. (See the COS Table Descriptions Internal Reference Manual, publication SM-0045, for a description of CNT.) The CNT contains one entry per device and one subentry for each path to that device.

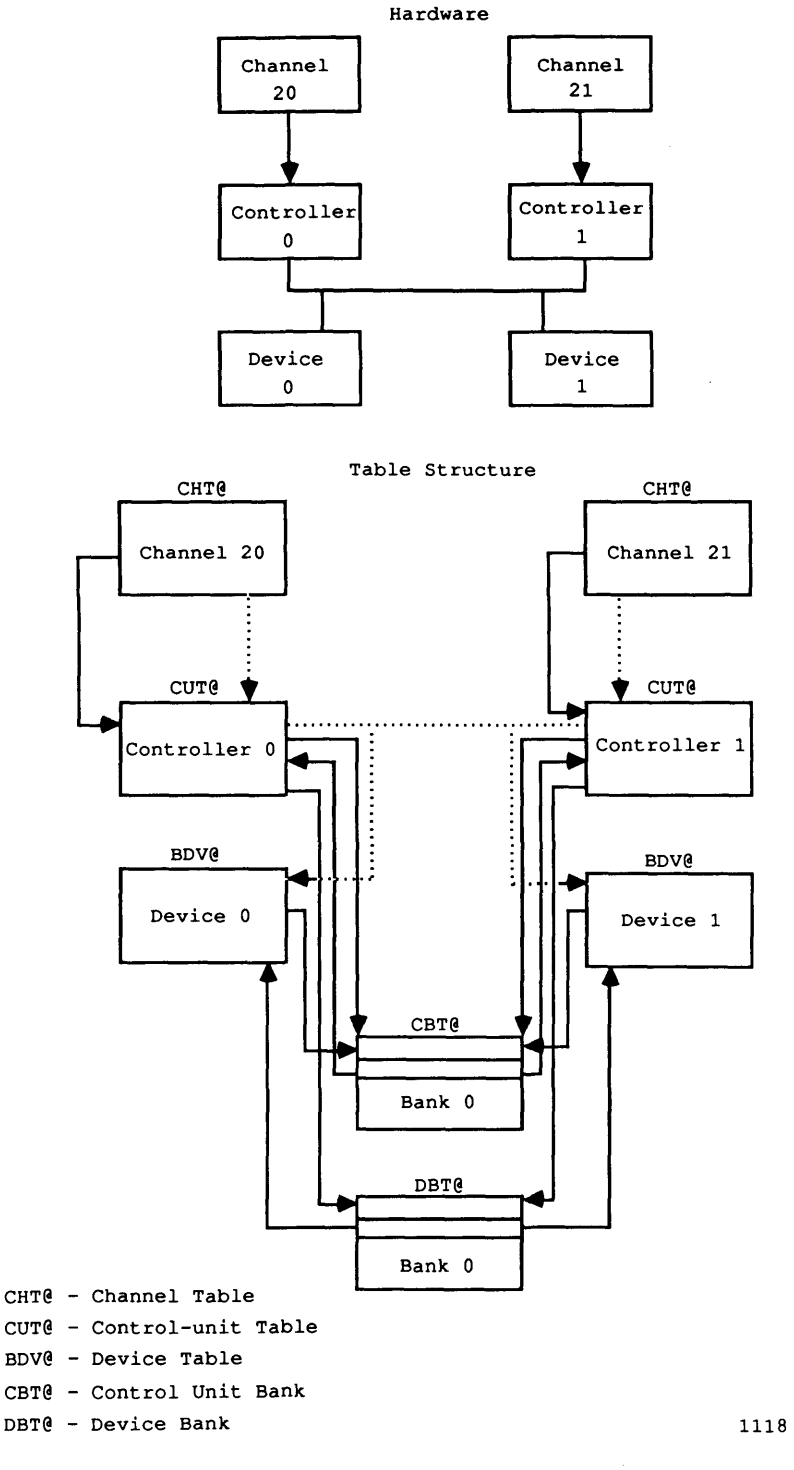


Figure 5-2. A 2-by-2 Configuration (Multiple Path, Single Bank)

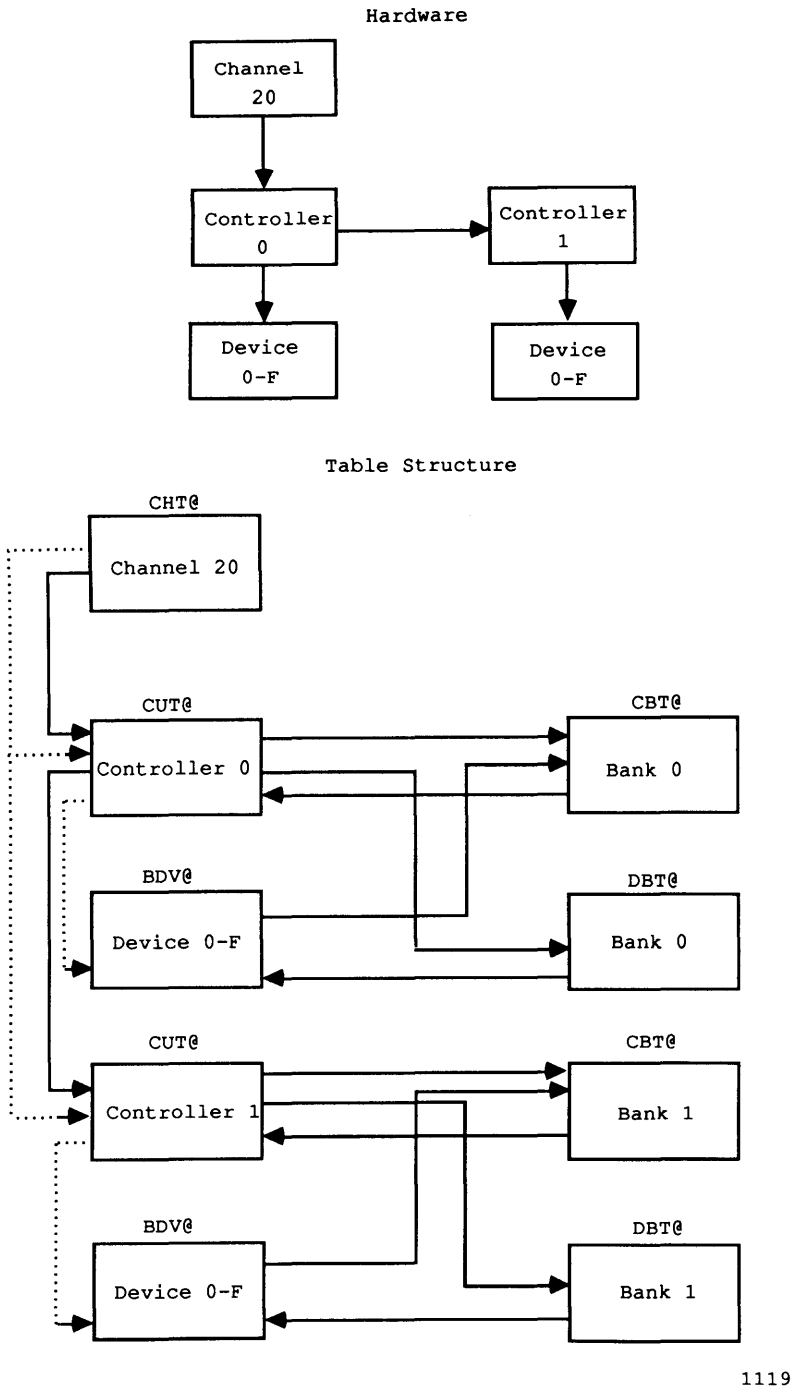


Figure 5-3. Two 1-by-1 Configurations (Single Path, Multiple Bank)

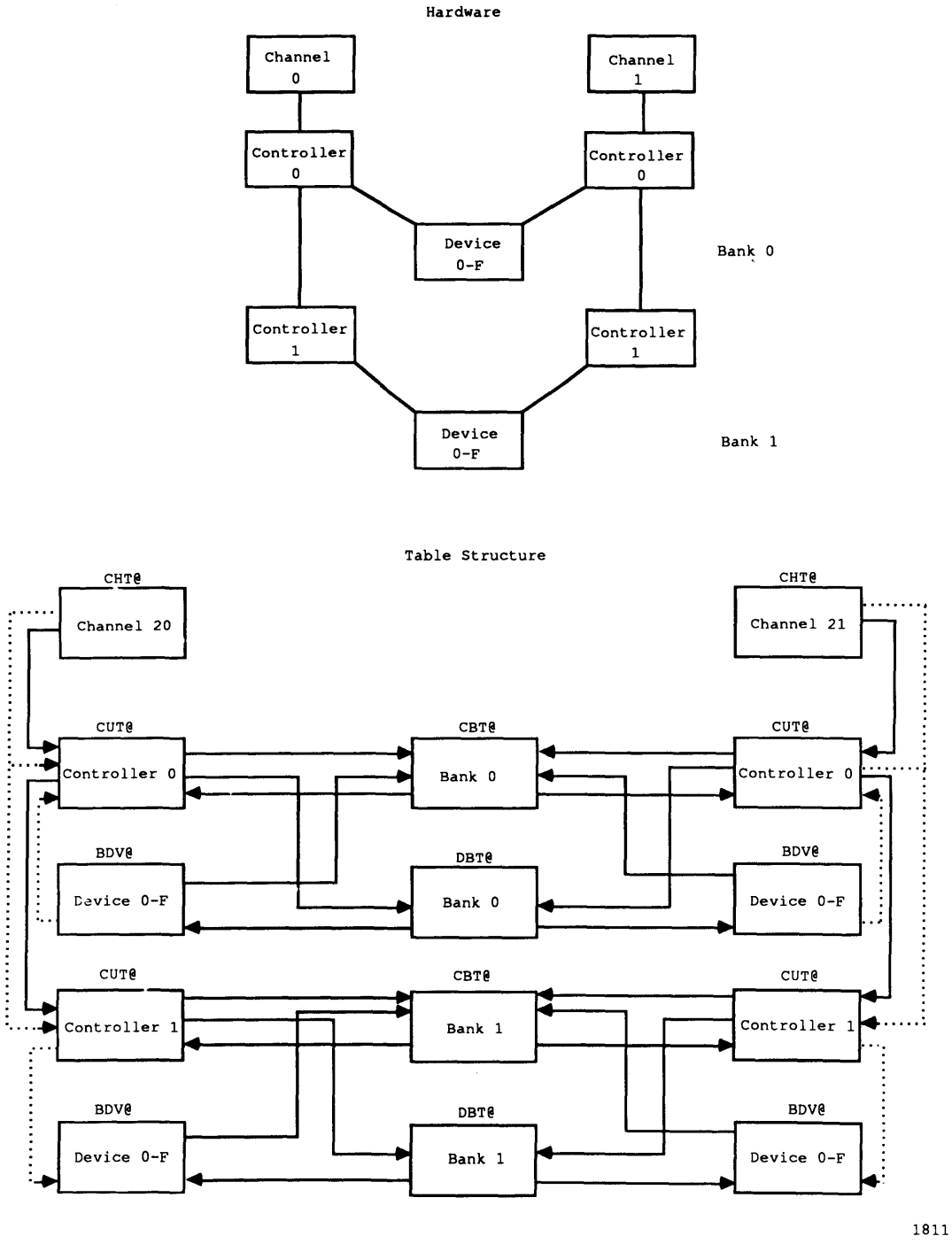


Figure 5-4. A 2-by-1 Configuration (Multiple Path, Multiple Bank)

Pointers to lists of tables in the Kernel table area are as follows:

Pointer	Description
XCHT	Points to a list of Channel Tables (CHT@) for all configured channels. The list is ordered by each channel's offset from channel 20_8 (see figure 5-5).
XDEV	Points to a list of Device Tables (BDV@) for all configured devices. The list is ordered by each device's logical ordinal number (see figure 5-6).
XCBT	Points to a list of Control-unit Bank Tables (CBT@) for all configured control units. The list is ordered by logical bank numbers.
XDBT	Points to a list of Device Bank Tables (DBT@) for all configured devices. The list is ordered by logical bank numbers.

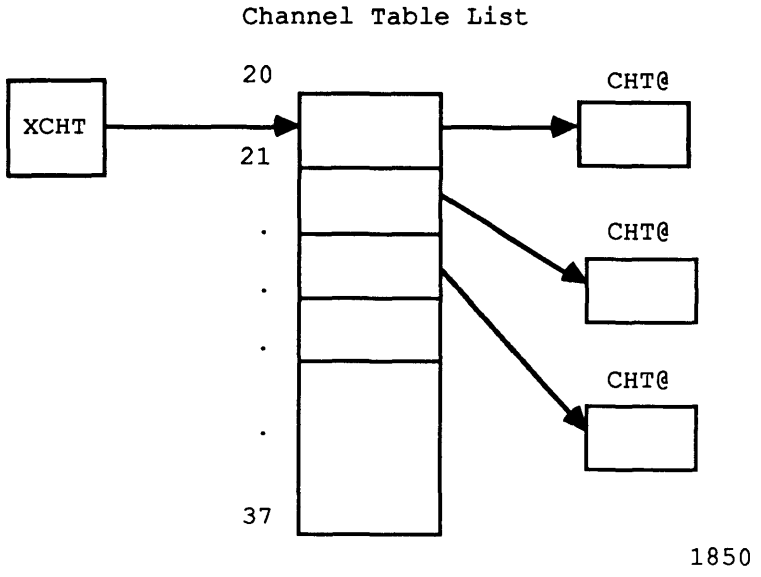


Figure 5-5. Pointer to Channel Tables for Each Configured Channel

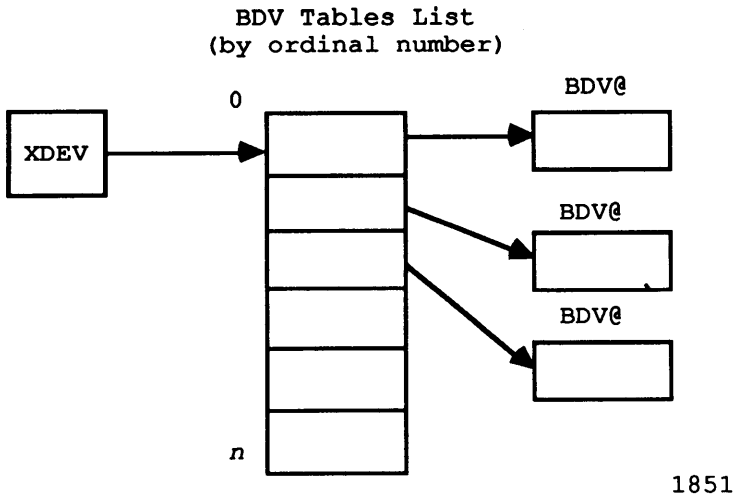


Figure 5-6. Pointer to Device Table for Each Configured Device

The following list briefly describes the BMX tables set up by BMXCPU (tables with a single use are described more fully here than the multipurpose tables). For field definitions and descriptions, see the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

Table	Description
BDV@	Device Table; each configured device has a BMX Device Table associated with it.
CBT@	Control-unit Bank Table; each configured control-unit bank has a Control-unit Bank Table used for path assignment. This table is composed of a header followed by a list of pointers to all Control Unit Tables (CUT@) included in the bank.
CHT@	Channel Table; each channel configured has a Channel Table associated with it.
CPB@	Command Parameter Block. Each open device has a Command Parameter Block associated with it. Where CPB@@LE specifies the number of parcels, the CPB is the first CPB@@LE parcels of the Tape Control Block allocated by a device activity when the device is opened.
CUT@	Control Unit Table; each configured control unit has a Control Unit Table.

DBT@ Device Bank Table; each Control-unit Bank Table has a corresponding Device Bank Table. This table is used by BMXDEM to find the Device Table corresponding to asynchronous (request-in) interrupts received. The table is composed of a header followed by a list of pointers to the Device Tables (BDV@) representing all devices in the bank.

5.4 CHANNEL PROGRAM WORD (CPW)

The CPW is the structure that conveys individual device commands from a device activity to the BMX software. CPWs are issued singularly or in lists, called command chains, and are classified into three types depending on the command requirements:

- Control commands that do not transfer any data to or from the device.
- Data transfer commands that use Local Memory only for data.
- Data transfer commands that use both Local Memory and Buffer Memory for data.

When a device activity calls the BMX device driver to issue commands, it stores the address of the first, or only, CPW in the Command Parameter Block (CPB) at CPB@CC.

CPW classification is accomplished by the device activity setting particular CPW flags. The following subsections describe these classifications and data required for each.

5.4.1 NONDATA TRANSFER COMMANDS

Nondata transfer commands are all control commands (although some control commands require data transfer, such as Load Display). The only data required in the CPW for these commands is the channel command and the flag CPW@DT set to zero. Command chaining is supported for these commands but is not used.

5.4.2 LOCAL MEMORY DATA TRANSFER COMMANDS

Local Memory data transfer commands transfer data to or from the device, but the data is not copied to or from Buffer Memory. The only two commands currently in this class are the Sense and Load Display commands. For these commands the following data is required:

- Field CPW@CM must contain the channel command.
- Flag CPW@DT must be set to 1 (indicating the command transfers data).
- Flag CPW@IN must be set to 1 for transfers from (Sense) or 0 for transfers to (Load Display) the device.
- Flag CPW@MS must be set to 0 (indicating Buffer Memory, MOS, is not used).
- Field CPW@DA must contain the address of the Local Memory data buffer (must be aligned on a word boundary).
- Field CPW@BL must contain the number of bytes to transfer. If the command transfers from the device, upon completion this field will contain the number of bytes actually transferred.

Command chaining is supported for these commands but is not used.

5.4.3 BUFFER MEMORY DATA TRANSFER COMMANDS

Buffer Memory data transfer commands are used to transfer tape blocks to or from the device. For reads (Read Forward and Read Backward), data transferred into Local Memory from the device is copied to preallocated Buffer Memory data buffers. For writes (Write), data is loaded from prefilled Buffer Memory data buffers into Local Memory for transfer to the device. The BMX software transfers the data between Local Memory and Buffer Memory as the data is being transferred to or from the device. A double-buffering scheme is used.

For these commands, the following data is required in the CPW:

- Field CPW@CM must contain the channel command.
- Flag CPW@DT must be set to 1 (indicating the command transfers data).
- Flag CPW@IN must be set to 1 for transfers from the device and set to 0 for transfers to the device.

- Flag CPW@MS must be set to 1 (indicating Buffer Memory, MOS, is to be used).

- For reads, fields CPW@BU and CPW@BL must be 0. When the command is complete, these two fields will contain the number of bytes actually transferred.

- For writes, fields CPW@BU and CPW@BL must contain the number of bytes to transfer to the device.

These commands also require the following data in the CPB:

- The DSC Buffer Descriptor Entry for the first sector of the first or only block must be placed in CPB@DE.

- The DSC's Buffer Memory address must be in CPB@DU and CPB@DL.

- The size of the DSC must be in CPB@LI.

- The offset of the above Buffer Descriptor Entry must be in CPB@PT.

- CPB@B0 and CPB@B1 must contain the addresses of two Local Memory data buffers.

- CPB@BP, CPB@0R, CPB@1R must be set properly for writes. Whichever buffer CPB@BP points to must be filled with the data for the first segment of the first block and its ready flag must be set (CPB@0R or CPB@1R).

- For reads, the maximum block size (CPB@MU and CPB@ML) must be set.

- Field CPB@PR must address an initialized PRW.

Command chaining by the TAPEIO overlay is usually used for these commands. Error Recovery never command chains its block I/O.

5.4.4 COMMAND CHAINING (CPW@CC)

Command chaining provides the most efficient method of issuing multiple commands to the same device. With command chaining, the BMX software does not reassign an I/O path for each command, thus minimizing channel and control unit overhead.

Command chaining requires an array, called a CPW list, of at least two CPWs. The boundaries of the list must be set in the fields CPB@CB and CPB@CE. Additionally, all CPWs but the last one to be processed must have the flag CPW@CC set to one.

The BMX software processes the list in a circular fashion. The first CPW executed is the one pointed to by CPB@CC, the next is the one with an address 4 parcels greater. List wrapping occurs when the CPW just completed has an address equal to CPB@CE; the CPW addressed by CPB@CB is then executed. The field CPB@CC is always updated to reflect the CPW currently being executed.

Command chaining continues until the CPW just completed has the CPW@CC flag set to 0, an exceptional status is presented by the device, or the BMX software detects some kind of overrun condition.

As each CPW is completed, the BMX software returns to the caller with an Operation Status (CPB@OS) equal to OS$BZ. This allows the caller to rebuild the CPW just processed. Specifically, the caller must clear the flag CPW@DN (which is set by the BMX software) as an indication that the caller's processing is in synchronization with the command chain. Additionally, the caller must not clear the flag CPW@CC in CPWs still pending execution (as opposed to those with the CPW@DN flag set), since, in certain time-dependent situations, the device may still be command chaining.

5.5 DESCRIPTION OF ROUTINES

The following subsections describe the BMX channel interface routines in detail.

5.5.1 BMXCON

BMXCON configures the subsystem components up or down.

Format:

Location	Result	Operand
	CALL	BMXCON,(*cm,ch,cu,dv,of,dw*)

cm Component:

CON$CHN	Configure channel
CON$CTU	Configure control unit
CON$DEV	Configure device

ch Channel number. Required if configuring either channel or control unit.

cu Control unit ID. Physical address. Required if configuring control unit.

dv Device ordinal. Logical device ordinal associated with the device. Required if configuring device.

of Component Off-line flag:

CON$ON	Configure specified component on-line (available)
CON$OFF	Configure specified component off-line (not available)

dw Device Down flag:

CON$UP	Configure device up
CON$DOWN	Configure device down

5.5.1.1 Channel configuration (CON$CHN)

If configuring the channel down, the flag CHT@OF is set in the Channel Table to indicate that the channel is configured down. All control units attached to the channel are marked in the associated Control Unit Tables as not available for assignment (CUT@NA). Current I/O is allowed to complete normally.

If configuring the channel up, the CHT@OF flag is cleared in the Channel Table and all attached control units are marked as available (CUT@NA clear). If the channel had previously been down, BMXCON issues a System Reset channel function and turns various data patterns around through the channel registers as a minimal checkout before configuring the channel up.

5.5.1.2 Control unit configuration (CON$CUT)

If configuring the control unit down, the flag CUT@OF in the Control Unit Table is set to indicate that the control unit is down. Current I/O is allowed to complete normally.

If configuring the control unit up, the CUT@OF flag in the Control Unit Table is cleared. If the control unit is not currently in use (CUT@CO clear) and a channel is on-line and free, a TEST-I/O command is issued. The resultant input tags are checked for select-in tag (IT$SLI). Select-in is set to indicate that the addressed control unit cannot be found and the control unit is then left down.

5.5.1.3 Device configuration (CON$DEV)

Unlike channels and control units there are two states defined for devices: on-line/off-line and up/down. A drive that is off-line is simply that. A drive that is on-line may be either up or down. All configuration requests contain values for both states.

If configuring the device off-line, the flags BDV@DW and BDV@OF in the Device Table are set (a drive cannot be both off-line and up). If configuring the device on-line, the flag BDV@OF is cleared. If configuring the device down, the flag BDV@DW is set.

When configuring the device up when it had previously been down, a check is made to determine if any paths are configured on-line; if not, the request is rejected. If at least one path is on-line, a Selective Reset is issued. The device is then queried for ready status, indicating the presence of a tape mounted and in ready state. If the device is found to be ready, a rewind-unload is issued. The flag BDV@DW is then cleared.

5.5.1.4 BMXCON messages

BMXCON displays a message to the Auxiliary I/O Processor (XIOP) console for each configuration request received. There are three types of messages corresponding to three component types: device, channel, and channel/control unit.

The information each message provides is as follows:

Information	Description
Time	Time of day that message issued
Component type	Channel, device, or channel/control unit and address of component
Status	Status of component. The Data Expected and Data Received information is part of the Data/Byte/Status register error information.

The format for each type of message follows. In these descriptions *x* is the physical address of the device or control unit in hexadecimal, *nn* is the bank number in octal, *oo* is the channel number in octal, and *llllll* and *mmmmmm* are octal values.

Device message format:

Time	Component Type	Status
time	device (x) bank (*nn*)	One of the following: -off-line -on-line/up -on-line/down -not available -not configured

Channel/control unit message format:

Time	Component Type	Status
time	channel(*oo*)/control unit(x)	One of the following: -off-line -on-line -not configured -not available -time-out

Channel message format:

Time	Component Type	Status
time	channel (*oo*)	One of the following: -off-line -on-line -not configured -not available -time-out and in addition, any of the following: -data register error -byte register error -status register error plus: - Data Expected: *llllll* - Data Received: *mmmmmm*

5.5.2 BMXCPU

BMXCPU allocates and builds all tables used by the BMX subsystem through information in the CNT received from the mainframe. See the COS Table Descriptions Internal Reference Manual, publication SM-0045, for more information on CNT.

Format:

Location	Result	Operand
	CALL	BMXCPU,(*dal*)

dal Address of DAL

BMXCPU first scans the CNT for the number of BMX device entries present. The CNT is then processed, one entry at a time. There is one subentry for each path available to the device. These subentries contain the necessary information to build the Channel (CHT@), the Control Unit (CUT@), the Bank (CBT@, DBT@) Tables, and the Device (BDV@) Tables.

5.5.3 BMXSIO

BMXSIO serves as the interface between each device activity and the BMX channel driver (BMXDEM). It is referred to as the BMX device driver.

Format:

Location	Result	Operand
	CALL	BMXSIO, (*fn,cpb,dvn*)

fn Function code:
RQ$SIO Start I/O to a device
RQ$WIO Wait for I/O to complete on a device
All other values are passed to BMXAIO (this interface is not currently used).

cpb CPB address. The CPB is the BMX communication area for the device activity. (See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for more information.)

dvn Device ordinal. Logical device address.

5.5.3.1 Start I/O (RQ$SIO)

The Start I/O sequence is requested by the device activity to issue I/O to a BMX device.

Assign device path - BMXSIO begins the Start I/O sequence by assigning to the device a path composed of a channel/control-unit pair.

BMXSIO accomplishes path assignment through the use of the Control-unit Bank Table (CBT@) associated with the device. This table is found in the Device Table (BDV@CB) associated with the device.

The Control-unit Bank Table (CBT@) contains a list of pointers to all Control-unit Tables (CUT@) for the control units that have access to the device. The search begins with the value of pointer BDV@LC as the most likely path to be available, since it was the last one used.

BMXSIO searches the Control-unit Bank Table list for a control unit that is both free (CUT@CO = 0) and available (CUT@FL = 0). If no control units are available, the device activity is suspended by a PUSH onto the control-unit bank queue (CBT@QU). The control-unit bank queue is serviced by each device activity when the path assignment for the device is released.

When a control unit is found, the channel number in the Control Unit Table (CUT@CN) is used to locate the attached channel. If the channel is available, indicated by a zero in field CHT@CO, that channel/control unit path is selected. If the channel is not available, the device activity is suspended by a PUSH onto the channel resource queue (CHT@QU). The channel resource queue is also serviced by each device activity when the path assignment for the device is released. Also, it is serviced by asynchronous (Request-in) interrupt processing of the BMX channel driver. Once the device activity is resumed from the channel resource queue, device path assignment starts all over again (by searching for the next available control unit). This is necessary because the available control unit and/or channel may have been assigned by another device activity while this one was suspended.

When a control unit and channel are found, the Device Table address is entered in the Control Unit Table (CUT@CO), which marks the control unit as assigned. The current pointer into the Bank Table (CBT@) is saved in the Device Table (BDV@LC) for subsequent assignments. Entry of the Control Unit Table address in the Channel Table (CHT@CO) assigns the channel.

Finally, the physical path address is formed by combining the control-unit address (CUT@CA) with the device address (BDV@UN) and storing it back in the Device Table (BDV@UN). This address, along with the channel number, is used by the driver to issue I/O instructions to the device.

There are three special cases related to device path assignment as follows:

- Contingent connection

  A device error is detected by the presence of unit check (ST$UC) in the device status (BDV@DS). BMXSIO sets the Error flag (BDV@ER) to indicate that a contingent connection exists.

  When a unit check is received in the ending status, a contingent connection exists between the control unit and the device. This condition remains in effect until sense information related to the unit check is taken from the control unit. BMXSIO detects this condition by reading the flag BDV@ER from the Device Table. If the flag is set, BMXSIO only assigns the last path used (BDV@LC) to the device. In addition, BMXSIO validates that the command in the CPW to be issued is a Sense I/O (CM$SNS) command.

- Control-unit assign

  BMXSIO detects control-unit assign when either of the CPB flags CPB@PA or CPB@TA are set, which indicate that the last path used by the device (BDV@LC) is to be assigned. The flags are set by the device activity, either for on-line diagnostics (CPB@PA) or error recovery (CPB@TA). The flags remain set until cleared by the calling routine.

- Path already assigned

  It is possible, because of time-dependent situations involved with asynchronous (Request-in) processing, that a path is already assigned to the device. In this case, that path is reused.

Once a device path has been determined, the request is sent to BMXDEM for processing. This is accomplished by entering the Start Command sequence (KIC$SC) into the assigned Channel Table (CHT@NP) and placing the Channel Table on a queue (XCIQ) for BMXDEM. BMXDEM is then activated (see BMXDEM in this section).

BMXSIO suspends the device activity on the Device Table queue (BDV@TQ) and remains there until activated by BMXDEM or as a result of a device time-out.

Device time-out - Usually BMXSIO uses the TPUSH (timed push) mechanism to suspend the device activity on the device queue. If control is returned to BMXSIO with the time-out count exhausted (BDV$TMO), an error (ES$DTO) is put into the CPB (CPB@EC) and an error operation status (OS$ER) is returned to the caller.

BMXSIO uses the PUSH (untimed push) mechanism to suspend the device activity if the Device Not-ready flag (BDV@NR) is set. This flag is set by the device driver when an interrupt is expected only after manual intervention such as a tape being mounted.

Control-unit busy - Control-unit busy is indicated by the presence of Busy (ST$BZ) along with Status Modifier (ST$MD) in the device status (BDV@DS). Control-unit busy means that the selected control unit is not in a state where it can accept commands from the attached channel. BMXSIO marks the control-unit busy in the Control Unit Table (CUT@BZ) and selects a new path for the device. The I/O request is then reissued to BMXDEM. The Control-unit Busy flag is cleared when the control unit presents control unit end status (ST$CUE) through asynchronous (Request-in) interrupt.

Device busy - Device busy is indicated by the presence of Busy (ST$BZ) without Status Modifier (ST$MD) in the device status (BDV@DS). Device busy means that the selected device is not in a state where it can accept commands from the selected control unit. The defined BMX channel protocol requires the device to present an asynchronous (Request-in) interrupt with a Device End (ST$DE) in the device status (BDV@DS). BMXSIO marks the Device Table as pending CPW restart from a busy (BDV@RS), waiting Device End (BDV@WE) and Request-in (BDV@RI), and suspends the device activity on the device queue. When the Request-in interrupt is received from the device, BMXDEM will resume the device activity and BMXSIO reissues the request to BMXDEM.

Channel Command Retry - Channel Command Retry is indicated by the presence of Status Modifier (ST$MD), Channel End (ST$CE), and Unit Check (ST$UC) in the device status (BDV@DS). Channel Command Retry means that the selected control unit must perform some internal operation before the command can be accepted. The device status may contain Device End (ST$DE) also. If it does, the command may be reissued immediately; if it does not, the defined BMX channel protocol requires the device to present an asynchronous (Request-in) interrupt with a Device End (ST$DE) in the device status (BDV@DS).

In the case where Device End is not present, BMXSIO marks the Device Table as waiting Device End (BDV@WE) and Request-in (BDV@RI) and suspends the device activity on the device queue. When the Request-in interrupt is received from the device, BMXDEM will resume the device activity and BMXSIO then returns to the caller with an Operation Status of OS$RT.

In the case where Device End is present, BMXSIO immediately returns to the caller with an Operation Status of OS$RT.

5.5.3.2 Wait I/O (RQ$WIO)

The wait I/O (RQ$WIO) request is made by the device driver when awaiting ending status from a device used in a previously initiated I/O operation or for another command to complete when command chaining. BMXSIO determines whether the device is busy (OS$BZ or OS$IP). If it is, BMXSIO suspends the activity on the device wait queue (BDV@TQ).

When reactivated after suspension on the device wait queue, BMXSIO determines the current status and takes appropriate action.

5.5.3.3 Return to caller

Before returning to the caller, BMXSIO determines whether or not to release the device path. The device path is released if ending status has been received and if no error has occurred.

Releasing the device path consists of clearing channel ownership (CHT@CO) and control unit ownership (CUT@CO), enabling request-in interrupts, and servicing the Control Unit Block Queue (CBT@QU) and the Channel Resource Queue (CHT@QU). Request-in interrupts always remain enabled when the device path is not assigned.

5.5.4 BMXAIO

BMXAIO handles auxiliary functions for the BMX subsystem. These are functions that are not frequently used.

Format:

Location	Result	Operand
	CALL	BMXAIO, (fn,p_0,p_1,p_2)

fn Function code:
- RQ$HIO Halt I/O
- RQ$APTH Assign Path (DIA task only)
- RQ$RPTH Release Path (DIA task only)
- RQ$RSET Reset channel/device

p_0 CPB address if *fn* = RQ$HIO or RQ$RSET
Channel number if *fn* = RQ$APTH or RQ$RPTH

p_1 Device ordinal

p_2 Control unit address if *fn* = RQ$APTH or RQ$RPTH
BMA:1 accumulator value (reset type) if *fn* = RQ$RSET

5.5.4.1 Halt I/O (RQ$HIO)

This function requests that BMXAIO terminate I/O to the device indicated by the CPB address and device ordinal.

If the channel (CPB@CN) is currently busy, BMXAIO issues an interface disconnect to the channel, terminating all I/O to the device and freeing the device path.

BMXAIO then releases the device path, which involves clearing CHT@CO, clearing CUT@CO, and enabling request-in interrupts. A status of OS$HD is returned to the caller.

5.5.4.2 Assign device path (RQ$APTH)

This function is used by on-line BMX diagnostics to assign a device path and allows a diagnostic to do its own I/O concurrent with the system.

The on-line diagnostic request DIA sets a diagnostic request bit in the packet. BMXAIO uses logic similar to BMXSIO to assign the path (see above).

BMXAIO returns a status (0) indicating successful assignment; otherwise a protocol error is returned.

5.5.4.3 Release device path (RQ$RPTH)

This call is made to release a path previously assigned by the RQ$RPTH request.

BMXAIO releases the device path by clearing all previously assigned tables, reinitializes the channel, and returns to the caller. If the device is not active, a protocol error is returned.

5.5.4.4 Request reset (RQ$RSET)

BMXAIO finds a path and issues a selective reset or channel reset depending on the caller's parameters.

5.5.5 BMXDEM

BMXDEM is the BMX subsystem channel driver. It is responsible for the initiation and control of all I/O to a BMX device through the assigned channel and control unit.

BMXDEM receives all of its requests from the channel queue (XCIQ), where the address of the Channel Table (CHT@) corresponding to the assigned channel is placed.

BMXDEM is activated by BMXSIO, to initiate I/O, and the BMX channel interrupt handler, IBMX, which processes interrupts.

BMXDEM is driven by the sequence code stored in the Channel Table (CHT@NP). All necessary tables can be located by BMXDEM using the Channel Table, as shown in figure 5-7.

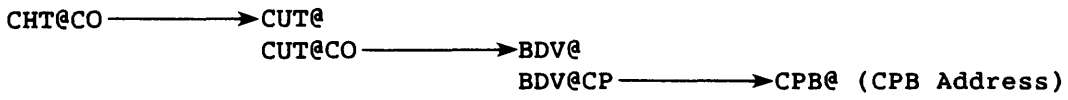


Figure 5-7. BMXDEM's Usage of the Channel Table

5.5.5.1 Start command sequence (KIC$SC)

The Start command sequence initiates I/O to the specified device (BDV@UN), beginning with the CPW addressed by CPB@CC.

BMXDEM determines from the command (CPW@DT) whether or not a data transfer is indicated. If a data transfer is indicated, the following additional processing is done.

Data transfer - To accomplish a data transfer, the channel data buffer and byte-count registers must be set. The channel hardware contains two data-buffer and two byte-count registers to provide data chaining capability. Data chaining allows for data transfers too large to be contained in a single Local Memory data buffer and is used only if Buffer Memory is to be used for the command (CPW@MS set to one).

The data address word entered into the channel data address registers contains two flags, in the low-order 2 bits, which control hardware data chaining. The format for the word is as follows:

Buffer Address	DC	BP
2^{15}	2^{2}	2^{1} 2^{0}

Flag	Description
DC	Data chain; if this bit is set, the channel hardware automatically switches to the buffer address in the other buffer register. The switch occurs when the byte counter decrements to 0. The byte-count register is updated from the auxiliary byte-count register. The hardware returns an

DC (continued)	interrupt when the switch occurs to allow the software to process the completed buffer and reset the buffer address register and byte count if the data chain continues.
BP	Buffer register pointer; indicates which of the two registers the buffer address applies to.

If data chaining will not be used for the command, BMXDEM sets both buffer addresses and the first byte count from the current CPW (CPW@DA, CPW@BL). The second byte count is set to one.

If data chaining is to be used, the first address is set from whichever buffer pointer is pointed to by CPB@BP (CPB@B0 or CPB@B1). The second address is set to the other pointer. For transfers from the device, the two byte-count registers are set to 4096 bytes and the data chain bit is set in both buffer address registers. For transfers to the device, if the transfer length in the CPW (CPW@BU and CPW@BL) is less than 8192 bytes, then the two byte-count registers are set to the transfer length; otherwise, the two byte-count registers are set to 4096. The data chain bits in the buffer address registers are set appropriately.

Issuing the command - Before actually issuing the command, the device address/mode word must be sent to the channel. The format for this word is as follows:

Field	Description
CM	Channel mode. This field indicates which of the following interface protocols is to be used between the channel and the control unit: • DC Interlock or Offset Interlock • 4.5 MByte/second Data Streaming • 3.0 MByte/second Data Streaming BMXDEM obtains the value for this field from BDV@CM.
IM	Interrupt mode. This field indicates the conditions under which the hardware will generate an interrupt to the system. When issuing commands, this field is always set to zero, causing the hardware only to generate an interrupt when ending status is received or when the byte counter decrements to 0 during a data transfer.
CCM	Command chain mode. This field indicates the condition on which the control unit is to chain. BMXDEM sets command chain on Device End if indicated in the CPW (CPW@CC). This mode leaves the device connected to the device path after the current operation completes, if no abnormal status conditions exist. The next command can then be issued immediately.

Field	Description
STK	Stack Status; this flag tells the control unit to hold any pending status until the channel is ready for it. This flag is never set by BMXDEM.
SK	Skip Data Transfer; this flag tells the channel not to transfer any data on a data transfer. This flag is never set by BMXDEM.
DA	Device address; this field contains the physical address of the device. It consists of control-unit address in the high-order 4 bits (0 through F) and the device address (0 through F) in the low-order 4 bits. BMXDEM obtains the device address from the Device Table (BDV@UN) and then issues the command from the CPW (CPW@CM) to the channel.

If the command issued was a data transfer to the device, BMXDEM activates the data handler pointed to by the PRW address from the CPB (CPB@PR). The next Local Memory buffer can then be filled with data while the current one is being sent to the device.

Channel time-out - After issuing the command, BMXDEM calls the QTIME routine, with the channel timer address (CHT@TO) and timer value (CHT$TMO), for placement on the event timer queue. The time-out handler address (IBMXTO) is entered into TMR@RT of each channel's timer entry (CHT@TO) at initialization time.

If a channel time-out occurs (the count for a channel that is being timed decrements to 0), IBMXTO is activated. IBMXTO disables interrupts on the timed out channel, enters an error code (ES$CTO) in the Channel Table (CHT@EC), and activates BMXDEM. BMXDEM returns the error to the device activity via BMXSIO by resuming it.

Sequence code update - Finally, BMXDEM updates the sequence code. If the command was for a data transfer, the sequence code is set to Advance Data (KIC$AD). Otherwise, the code is set to Advance command (KIC$AC). The code is saved in the Channel Table (CHT@NP) and is processed on the next call to BMXDEM for the channel.

5.5.5.2 Advance command sequence (KIC$AC)

The Advance command sequence is entered after the completion of current I/O. Completion is indicated when the Channel Done flag is set.

BMXDEM always activates the device activity upon entering the Advance command sequence. BMXDEM then determines from the ending status (CHT@DS) whether or not to continue processing. Ending status can be as follows.

Interrupt pending - If Device End (ST$DE) is not set in the device status, BMXDEM sets the operational status (BDV@OS) to interrupt pending (OS$IP) and terminates processing on this device. Interrupt pending indicates that the paths (channel and control unit) to the device are free to be reassigned, but that the device is still busy processing the command. Device end is reported through the asynchronous (request-in) mechanism when the device has completed the command.

Unit check/unit exception - Unit check (ST$UC) or unit exception (ST$UE) in the device status indicates an abnormal condition. BMXDEM returns an error status (OS$ER) in the operational status word (BDV@OS) and terminates processing on the device.

If no abnormal condition exists, BMXDEM checks the CPW (CPW@CC) for command chaining. If set, BMXDEM advances to the next CPW in the list. Processing from this point is the same as for the Start command sequence described previously.

5.5.5.3 Advance data sequence (KIC$AD)

The Advance data sequence is used when processing a Data Transfer command and is responsible for sustaining the data transfer until ending status is received. Ending status is indicated if the Channel Done flag (CHT@DN) is set.

The Advance Data sequence is activated by the channel interrupt handler (IBMX) each time an interrupt is received, indicating that the channel has switched data buffers. The software can then process one buffer while data is transferring between the other buffer and the device.

BMXDEM determines the byte count transferred on the completed buffer and, if a transfer from the device, increments the count in the associated CPW (CPW@BU and/or CPW@BL). The data handler is then activated via the PRW pointed to by the CPB.

Once a data transfer begins, the interrupt handler and, on writes, the data handler, are responsible for sustaining the channel data transfer by setting new channel address and byte-count registers.

With data transfer still going, BMXDEM resets the channel time-out count and terminates. When the data transfer has completed, BMXDEM sets the sequence code to Advance command (KIC$AC) and processes that sequence.

5.5.5.4 Request-in sequence (KIC$ER)

The request-in sequence is used to process status presented asynchronously from the control unit or device. BMXDEM receives the ending status and device address from the interrupt handler (IBMX) in the Channel Table (CHT@DS and CHT@RA, respectively).

BMXDEM uses the device address to search for the Device Table associated with the device presenting status (see figure 5-8).

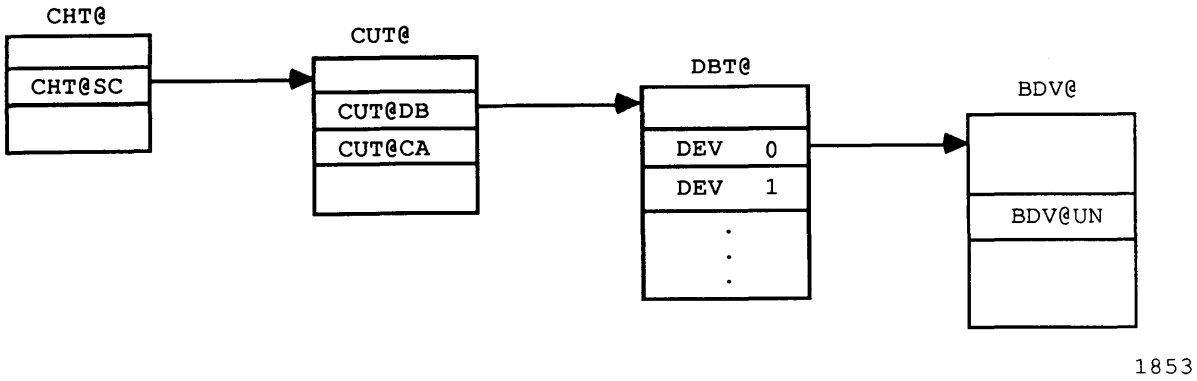


Figure 5-8. Location of BDV@UN

If the Device Table is found and the device is waiting for pending status (BDV@RI), BMXDEM resumes the waiting device activity (BDV@TQ). If the device is not waiting for pending status or if no Device Table exists for the device address presented, BMXDEM checks the ending status for control-unit end (ST$CUE). This condition indicates that the status is being presented as a result of a previous control-unit busy condition. BMXDEM clears control-unit busy (CUT@BZ) in the Control Unit Table and checks the control-unit bank queue (CBT@QU) for suspended device activities. If there are any, BMXDEM activates the first one queued.

BMXDEM locates the Device Table first by finding the Control Unit Table that has a control-unit address (CUT@CA) matching the one received. Once the Control Unit Table is found, the Device Table is found by indexing into the Device Bank Table (DBT@) associated with the control unit (CUT@DB). The index used is the device address within the field CHT@RA.

5.5.6 BMX INTERRUPT HANDLER (IBMX)

The BMX interrupt handler is responsible for handling all interrupts received on channels assigned to BMX interfaces. IBMX is activated by the Kernel interrupt handler using the Interrupt Jump Table (EITB).

IBMX disables interrupts for the channel and then locates the associated Channel Table (CHT@), using the BMX Channel Look-up Table (XCHT).

The channel timer entry (CHT@TO) is passed to DQTIME to remove the entry from the timer queue. Input tags, device status, and channel flags are saved in the Channel Table for use by the BMX driver (BMXDEM).

IBMX uses the sequence code (CHT@NP) to determine how to process the interrupt.

5.5.6.1 Immediate return (KIC$IR)

This sequence code indicates that the channel is assigned to an activity that is doing its own I/O (BMXAIO or BMXCON). IBMX activates the task (if there is one) waiting on the channel function queue (CHT@CF). No further processing is done for the interrupt.

5.5.6.2 Advance data (KIC$AD)

If the Channel Done flag is set, IBMX activates BMXDEM to process ending status on the data transfer.

If the Channel Done flag is not set, the channel has switched data buffers to sustain a chained data transfer. IBMX uses the data buffer pointer present in the channel input tags register to identify which Local Memory data buffer address (CPB@B0 or CPB@B1) to use to reset the channel in preparation for the next buffer switch. In the case of transfers from the device, data chaining is always indicated to the channel and a new byte-count register value of 4096 is set.

In the case of transfers to the device, data chaining is set only if the CPB indicates more than a sector remains to be transferred from Buffer Memory (fields CPB@BU and CPB@BL are greater than 4096). A new byte-count register value is never set, deferring that to the data handler. This ensures that the Local Memory buffer is reloaded before the channel switches to it.

IBMX checks the respective buffer ready flag in the CPB (CPB@0R or CPB@1R) to see if the new buffer is ready for I/O. If the flag is not set, a software overrun error (ES$SOR) is set into the Channel Table error code (CHT@EC). BMXDEM is activated to process the completed buffer of data and prepare for the next interrupt or to process the error.

5.5.6.3 Start request-in (KIC$SR)

IBMX marks the channel in use by entering the request-in process as the channel owner (CHT@CO = CO$RI). IBMX then issues a channel function to read in the device address associated with the request-in. The sequence code is set to continue request-in (KIC$CR) and IBMX terminates processing until the next channel interrupt occurs.

5.5.6.4 Continue request-in (KIC$CR)

The Continue Request-in sequence saves the device address received from the previous sequence in CHT@RA and issues a channel function to read in the status from the interrupting device. The sequence code (CHT@NP) is updated to End Request-in (KIC$ER), and IBMX terminates until the next interrupt occurs.

5.5.6.5 End request-in (KIC$ER)

The End Request-in sequence saves the device status in CHT@DS, activates BMXDEM, and terminates.

5.5.7 BMXOPE

BMXOPE is called to open or close a BMX device for a device activity. Processing is based on the function code (TQ@FCN) and the device ordinal (TQ@DVN).

Format:

Location	Result	Operand	Comment
	CALL	BMXOPE,(*dal*)	

dal Address of a formatted mainframe request

5.5.7.1 Open (FC$MOUNT/FC$REMOUNT)

BMXOPE opens a device by placing the Activity Descriptor (AD) address of the activity into the Device Table (BDV@AI) based on the device ordinal. If the device is already open, BMXOPE returns a protocol error (ST@DAL) in the packet (TQ@STS). BMXOPE then passes the request on to BMXTPO for device-dependent processing.

5.5.7.2 Close (FC$FREE)

BMXOPE closes a device by performing the following steps:

1. Issuing a Sense command
2. Clearing the operator display on cartridge-type devices
3. Issuing a selective reset

4. Clearing device ownership (BDV@AI)

5. Releasing the device table space (CPB/TCB) addressed by BDV@CP

6. If a DIA task, calling BMXAIO to release any path assignment (RQ$RPTH)

5.5.8 BMXTPO

BMXTPO is called to complete the open of a BMX tape device.

Format:

Location	Result	Operand
	CALL	BMXTPO,(*dal*,*dvn*,*fn*,*sid*,*pdv*)

dal Address of a formatted mainframe request.

dvn Device ordinal. Logical device address.

fn Function code:
FC$MOUNT Mount a tape on the device
FC$RMNT Remount a tape on a new device

sid Sender ID:
RQ$CPU Request is from mainframe.
Otherwise request is internal.

pdv Previous device ordinal. FC$RMNT only.

The difference between FC$MOUNT and FC$RMNT is that any data associated with the previous device is copied to the new one for remounts.

BMXTPO performs the following functions:

1. If the function is FC$MOUNT or FC$RMNT to a different device, a Tape Control Block (TCB) is allocated.

2. If a Datastream Control Table does not exists, DSCGET is called to allocate one.

3. If the request is from the DIA (BMX on-line diagnostics) task, BMXAIO is called to assign the path (RQ$APTH).

4. If the device is of cartridge type and there is a Display Control Byte (TQ@BCS) in the DAL, a Load Display command is issued to the device.

5. The device is then armed. This involves issuing a No-op command to the device. The only acceptable states for the drive are to be ready at load point or not ready (unloaded). If the device has a tape mounted but it is not at load point, it is unloaded and the arm is retried. If the device is not ready, the device activity waits for tape to be loaded.

6. If a tape is successfully loaded, control is either passed to TEX to process further mainframe requests, if the request is from the mainframe, or returned to the caller.

6. I/O SUBSYSTEM STATION

The I/O Subsystem (IOS) station is a collection of closely associated tasks executing in the Master I/O Processor (MIOP)† that provide operator command and display facilities and dataset staging capabilities independent of any front-end computers.

The close association among the tasks is maintained through communication areas and shared data areas allocated in Local Memory.

The station operates under the control of the Kernel. All station code exists in the form of overlays stored in Buffer Memory. The Kernel manages the scheduling of the station tasks and the loading of overlays into Local Memory. A station task interfaces with other tasks and the Kernel through the standard Kernel service requests.

6.1 STATION TASKS

Table 6-1 lists the station tasks and describes the function of each. Each task name is also the name of the initial overlay or the controlling overlay for that task.

The station is initiated by entering the STATION command at an MIOP Kernel console. Multiple stations may be executing provided that resources, including a dedicated console for each, are available.

The STATION command initiates one set of the console handling routines: KEYBD, CLI, and DISPLAY. A set of these routines is also initiated to handle each console added to the station through the CONSOLE command. The tasks terminate when an END command is issued at the corresponding console.

Only one PROTOCOL task can exist at a time. It is created by the LOGON command and endures as long as communications with the mainframe are maintained. Communications are terminated explicitly by the LOGOFF command or automatically if the task encounters communication errors.

† A task in the Buffer I/O Processor (BIOP) is required to move messages between Buffer Memory and Central Memory. This task is not unique to the station in that it services all active stations and concentrators. This task is discussed in the section describing the concentrator.

Table 6-1. Station Tasks

Task	Function
CLI	Interprets and executes the operator commands
DISPLAY	Formats the operator displays
KEYBD	Receives characters entered at the console keyboard
PROTOCOL	Manages communications between the station and the mainframe
STAGEIN	Stages a dataset from an IOS input device to the mainframe
STAGEOUT	Stages a dataset from the mainframe to an IOS output device

A STAGEIN task is started by the PROTOCOL task for each input dataset staging operation; a STAGEOUT task is started for each output dataset staging operation. The tasks terminate when the staging operation completes or is aborted. The number of staging tasks simultaneously active is governed by protocol parameters assembled in the STATINIT overlay: the maximum input stream count (IST), the maximum output stream count (OST), and the maximum active stream count (AST).

6.2 STATION STORAGE

The station tasks allocate storage in Local Memory and Buffer Memory. Some of these storage areas are accessible by more than one task; the area contains shared data and intertask communication areas.

Pointers to the shared memory are maintained in global registers, allowing access by called overlays without requiring that the address be passed as a parameter. Table 6-2 indicates the tasks that access each shared memory area. The IOS Table Descriptions Internal Reference Manual contains detailed descriptions of the shared memory areas.

Table 6-2. Shared Memory Access

Register	Task					
	KEYBD	CLI	DISPLAY	PROTOCOL	STAGEIN	STAGEOUT
%STAT		x	x	x	x	x
%CLI		x	x			
%STCON	x	x	x			
%PROT			x†	x	x†	x†

† The reference is through a parameter rather than a direct reference using the global register.

The shared memory areas are as follows:

Register	Area Name	Contents
%STAT	Station shared Local Memory (SS@)	Station parameters and queued dataset information
%CLI	Console support tasks shared memory	Console display parameters
%STCON	Console Driver Table (C$)	Console parameters
%PROT	PROTOCOL task Local Memory (PT@)	Addresses and I/O Stream Control Tables used by PROTOCOL task

All task interaction occurs using the shared memory areas. This interaction takes two forms:

- Modifying parameters used by another task. This may be viewed as indirect interaction, because it does not directly affect task scheduling.
- Interfacing with another task through the Kernel service calls PUSH, POP, and TPUSH, all of which require Local Memory queue cells.

The two types of interaction are often used in conjunction. For instance, the LOCK, UNLOCK, WATCH, and SIGNAL macros require both a data parcel and a queue used by one of the PUSH, POP, or TPUSH service calls.

The individual tasks also allocate unshared storage areas. The Local Memory stack areas reserved by the CLI, DISPLAY, and PROTOCOL tasks allow the tasks to allocate and release small, variable size buffers through the GETSTACK and FRESTACK macros, respectively. This mechanism guarantees that a memory buffer is available when required and eliminates the overhead involved in GETMEM and RELMEM Kernel service calls. However, a memory buffer large enough to contain all simultaneously allocated buffers (for that task) must be reserved for the stack.

A stack (see figure 6-1) consists of a Local Memory buffer and two global registers: %STACK, which points to the next available location, and %LIMIT, which points to the end of the stack buffer. %STACK and %LIMIT are initialized by the INSTACK macro after the Local Memory buffer has been reserved.

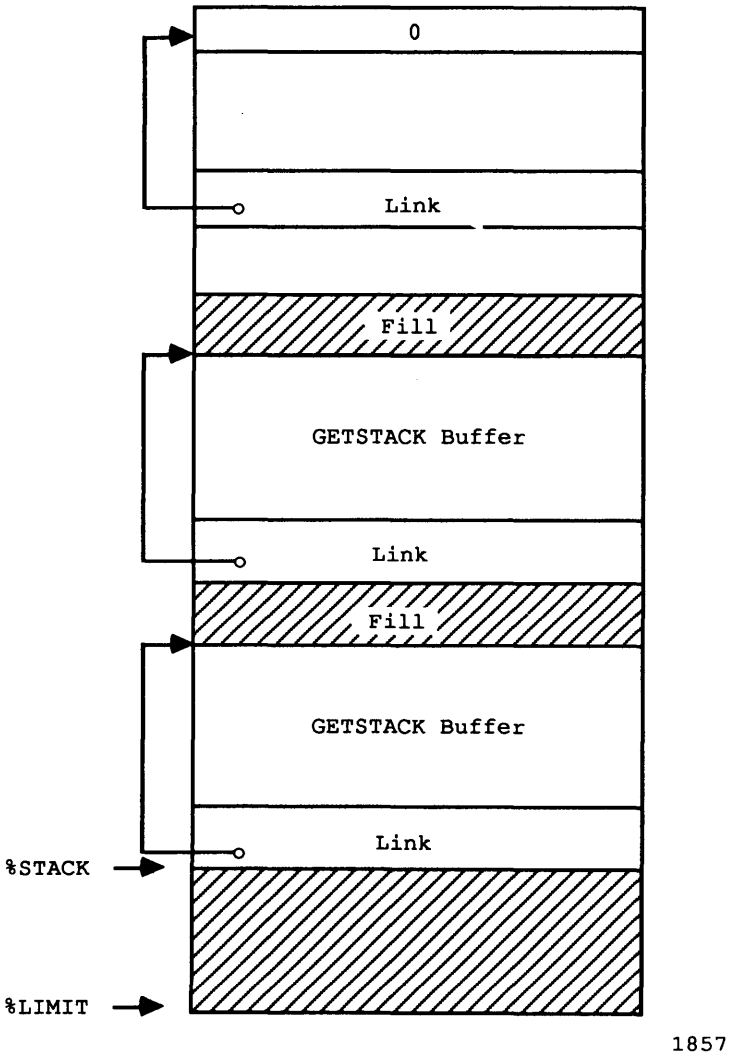


Figure 6-1. Local Memory Stack Area

Fill uses between 0 and 3 parcels, enabling all buffers to begin on a word boundary.

6.3 TASK FLOW AND INTERACTION

This subsection describes the general flow of the station tasks and the interaction between them. In the task flow descriptions, the overlays responsible for the function cited are listed on the right. Diagrams show the hierarchy of the overlays and the areas of interaction with the other tasks.

6.3.1 STATION INITIALIZATION

The station is initiated with the STATION command, which is entered at an MIOP Kernel console. The station initialization routines allocate and initialize buffers and create the triad of tasks KEYBD, CLI, and DISPLAY. If any aspect of initialization fails, all resources are released, and an explanatory error message is generated.

Station initialization is shown in figure 6-2. Its flow is as follows:

Step	Function	Overlay
1.	Validate the station console number	STATION
2.	Allocate and initialize shared Local Memory buffer (%STAT) and a shared buffer in Buffer Memory	STATINIT
3.	Usurp station console and allocate console support buffer (%STCON)	USURP ICONSL
4.	Allocate and initialize shared memory buffer (%CLI)	CLINIT
5.	Set up stack (%STACK and %LIMIT) and create DISPLAY task	CLINIT
6.	Create KEYBD task	CLINIT
7.	Write title line to console	CONSL
8.	Set up stack and create CLI task	CLINIT
9.	Output error message (if necessary)	STATION

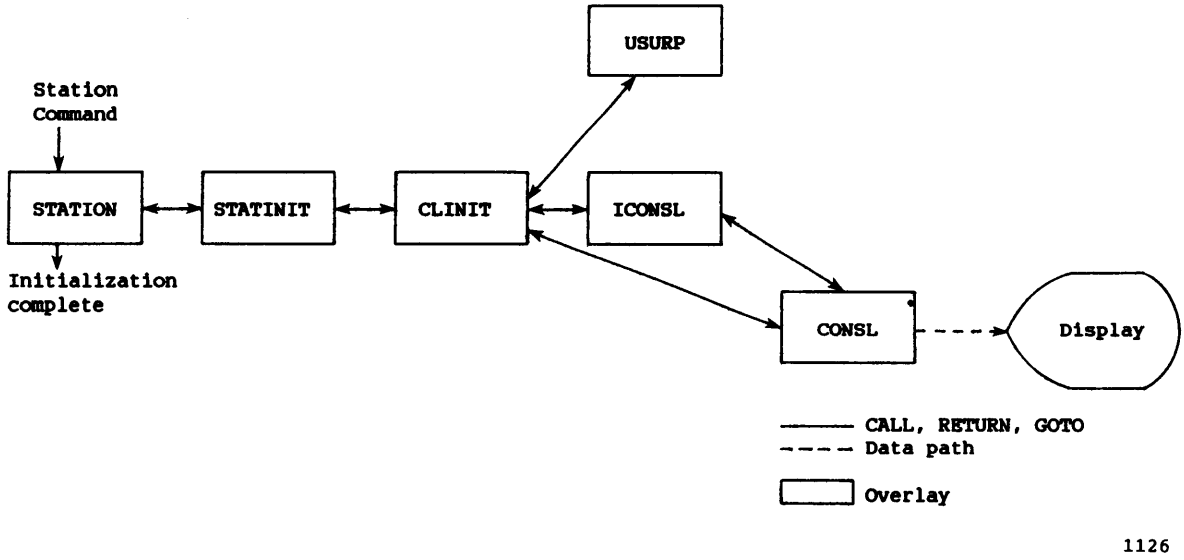


Figure 6-2. Station Initialization Flow

6.3.2 KEYBD TASK

The KEYBD task receives input from the console keyboard and passes it to the CLI task for interpretation. Because the character processing is negligible, the task is always available to receive input. Thus, the operator is able to type ahead; that is, key in commands before CLI has completed processing the previous command. The interaction areas for KEYBD are described in table 6-3.

The KEYBD task is shown in figure 6-3. Its task flow is as follows:

Step	Function	Overlay
1.	Check for termination request. If posted, respond to the CLI task and terminate. The KEYBD task is not responsible for releasing any resources.	KEYBD
2.	Receive next character	KEYBD
3.	Translate the character using the table appropriate to the console type. Characters requiring special handling by CLI (such as the carriage return) are translated to special codes. If the character is illegal, ring the console bell.	KEYBD CONSL
4.	Store character in the circular buffer and activate the CLI task (if necessary)	KEYBD

Table 6-3. KEYBD Task Interaction Areas

Register	Field or Table	Use
%CLI	$B@CB	Keyboard input circular buffer. The KEYBD task stores characters and updates the *in* pointer; the CLI task removes characters and updates the *out* pointer.
%CLI	$R@CLI	CLI activation area. When CLI is idle waiting for the next command to be entered, it suspends by watching this area (through the WATCH macro). KEYBD, upon receiving input, activates CLI with the SIGNAL macro.
%CLI	$T@KEY	Termination communications area. The KEYBD task monitors this area for a termination request. CLI, as part of END command processing, posts a message then suspends (using the same area), awaiting an acknowledgment from the KEYBD task.
%STCON	C$LOCK	Interlock that controls access to the display. This is used when the KEYBD task must ring the console bell.

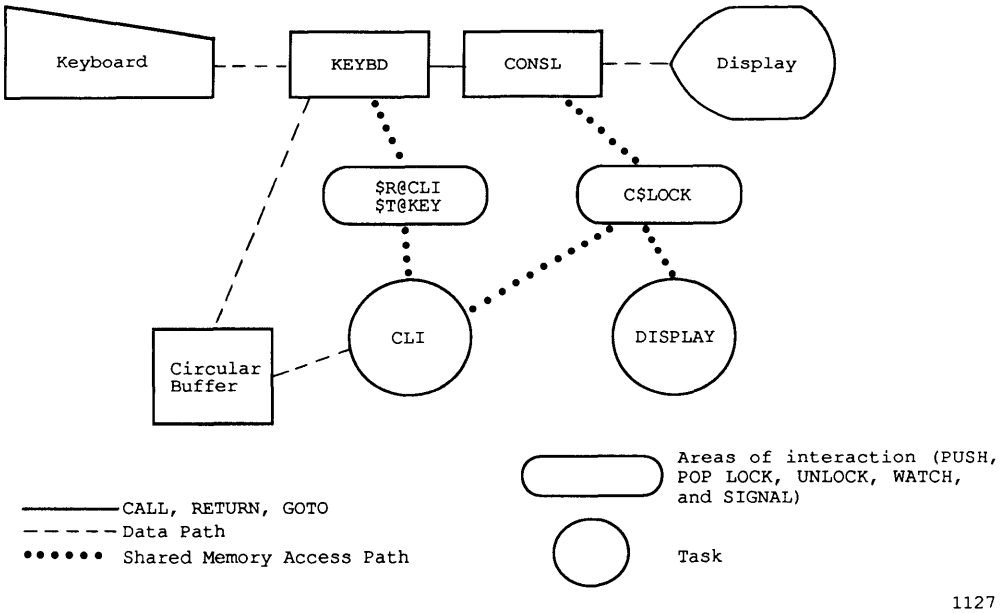


Figure 6-3. KEYBD Task Flow and Interaction

6.3.3 DISPLAY TASK

The DISPLAY task generates the operator and debug displays. It is responsible for acquiring and formatting the data and sending it to the console. The DISPLAY task execution is based on parameters stored in shared areas. These parameters, indicating display type, refresh rate, and so on, are set by the CLI task and are described in the task interaction areas table.

The DISPLAY task also interacts with other tasks and with the Kernel in the sense that it taps various tables for data used in the displays. The partial list of tables in table 6-4 includes shared memory areas (%STAT and %CLI), the I/O stream control tables, the expander device control tables, and the Kernel error logging table.

Table 6-4. DISPLAY Task Interaction Areas

Register	Field or Table	Use
%CLI	$R@DIS	DISPLAY task activation area. When the task is idle, it suspends the use of this area through a timed WATCH request. The task is reactivated either when the timer (the display refresh interval) expires or when CLI signals a display refresh due to a change of state (new display, LOGON or LOGOFF, and so on).
%CLI	$T@DIS	Termination communications area. See KEYBD task $T@KEY.
%CLI	$L@DIS	Interlock controlling access to the display parameters that follow. Its use prohibits the CLI task from altering the display parameters while the DISPLAY task is generating a display.
%CLI		The following are parameters controlling the DISPLAY task and describing the active display:
	$D@INT	Interval between display refreshes, in tenths of a second, if automatic refresh is enabled
	$F@FLG	Flag bit FLG$REF indicates whether automatic refresh is enabled.

Table 6-4. DISPLAY Task Interaction Areas (continued)

Register	Field or Table	Use
	$D@TYP	Display type: none, operator, or debug
	$D@OVL	Number of overlay controlling display generation: DISP01 Operator displays DISP02 Debug displays
	$D@PAR	Parameter defining operator display type (for instance, STATUS). The parameters for the various displays (DP$*type*) are generated by the DISPARS macro.
	$D@FRM	Frame number for the LINK, STATUS, and STORAGE displays
	$D@QUE	Queue flags for the STATUS display
	$D@DEB	Display descriptors: one descriptor for each debug display (A-Z)
	$D@LFT	Address of the debug display descriptor for the left debug display
	$D@RGT	Address of the debug display descriptor for the right debug display. If the right display is inactive, the address is 0.
	$D@MOD	Mode of the debug display (for instance, COS EXEC) if the mode is not defined in the display descriptor
	$D@TSK	COS task number used if the display mode is TASK and the number is not specified in the display descriptor
	$D@JOB	COS job sequence number used if the display mode is JOB and the JSQ is not specified in the display descriptor
%STCON	C$LOCK	Interlock controlling access to the console

Table 6-4. DISPLAY Task Interaction Areas (continued)

Register	Field or Table	Use
%STAT	SS@OP	Interlock controlling access to the Operator Stream Control Table
%STAT	SS@TAB	Operator Stream Control Table; it exchanges message request and response information between the DISPLAY and PROTOCOL tasks.
%STAT	SS@REQ	PROTOCOL task activation word. PROTOCOL task is activated, if necessary, after message information is stored in the Operator Stream Control Table.

The DISPLAY task flow and interaction on operator displays are illustrated in figure 6-4. The following steps are performed by the task:

Step	Function	Overlay
1.	Suspend, waiting for a display refresh request or the expiration of the automatic refresh interval.	DISPLAY
2.	Check for termination request; if posted, respond to the CLI task and terminate. The DISPLAY task is not responsible for releasing any resources.	DISPLAY
3.	Set the display interlock ($L@DIS).	DISPLAY
4.	If a display is active, do the following:	
	a. Toggle the display refresh indicator.	DISPLAY, CONSL
	b. Call DISP01 to process an operator display.	DISP01
	c. Call DISP02 to process a debug display.	DISP02
5.	Clear the display interlock.	DISPLAY

The DISP01 routine acquires the data for a display and calls an overlay to perform the formatting. In the cases of the LINK, STATUS, and STORAGE displays, the information is acquired from the mainframe through normal protocol messages. The DISPLAY task calls the POST overlay to send a message to the mainframe and to receive a response message. In fact, this overlay interfaces with the PROTOCOL task, which controls all protocol messages.

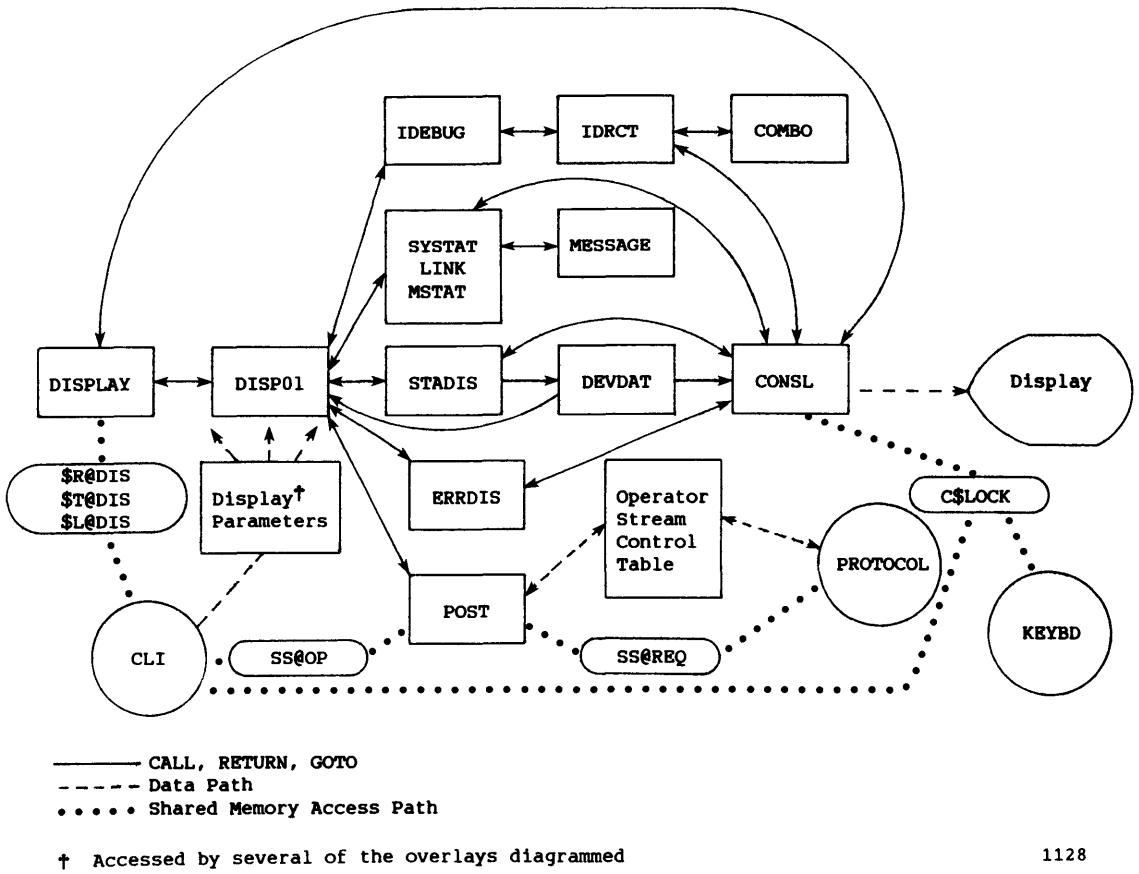


Figure 6-4. DISPLAY Task Flow and Interaction Operator Displays†

6.3.4 CLI TASK

The CLI task manages the operator interface; it interprets, executes, and responds to the station commands.

Other tasks can be directly involved in the processing of a command. For example, the commands that require communications with COS involve the PROTOCOL task. The END command requires termination processing by the KEYBD and DISPLAY tasks and possibly the PROTOCOL task (and thus, the STAGEIN and STAGEOUT tasks).

† Does not include MULTIPLY, DIVIDE, BTO, and BTD overlays

The tasks can also be affected by a particular command. For example, the REFRESH and LINK commands affect the DISPLAY task environment. The SAVE and SUBMIT commands queue a dataset for staging to the mainframe, impacting the PROTOCOL task and implying the creation of a STAGEIN task.

The validation of the command keyword is performed by the overlay CLI. Once the command type is established, an overlay is called to validate the remaining input parameters and execute the command. The overlays used to process commands have names of the form COMM*nn*; *n* is a decimal digit. The association between command keyword and the proper COMM*nn* overlay is made using the parameters CV$*command*, a unique index associated with the command. CV$*command* and CP$*command* are external symbols defined in the OVLNUM overlay using the COMPARS macro and referenced in the CLI overlay using the CMD macro.

The general flow of the CLI task follows. Examples of the processing of several commands are also included. The STATUS command is representative of the display commands, while the DROP command is typical of those requiring communication with COS. Except where indicated on figure 6-5, the areas of interaction are not enumerated; see the KEYBD, DISPLAY, and PROTOCOL task descriptions in this section. The task flow for CLI is as follows:

Step	Function	Overlay
1.	If reading from the console, input the characters from the keyboard circular buffer. Echo the character to the console or perform editing functions as required. Continue until an entire station command is accumulated.	READ
	If reading commands from a command file, read the next command line and display it on the console screen.	CFREAD
2.	Validate the command. A valid command is indicated by a special initial character or command keyword delimiter (for instance, + to roll the display) or by a command keyword that matches an entry in the command table.	CLI
3.	If the command is not recognized, output an error message and go to step 1.	ERROR MESSAGE CONSL
4.	Call the overlay that processes the given command.	COMM*nn*

For commands that may have parameters, perform steps 5 through 8.

Step	Function	Overlay
5.	Locate the table that describes the parameters.	DESCRIBE
6.	Build a table of information describing the parameters. Validate them using entries in the parameter descriptor tables. Convert parameters to an internal format if required (for instance, decimal to binary conversion).	DECODE DECOD2
7.	If a required parameter is invalid or missing, output an error message and return an error response.	ERROR MESSAGE CONSL
8.	Perform remainder of processing for the particular command.	COMM*nn*

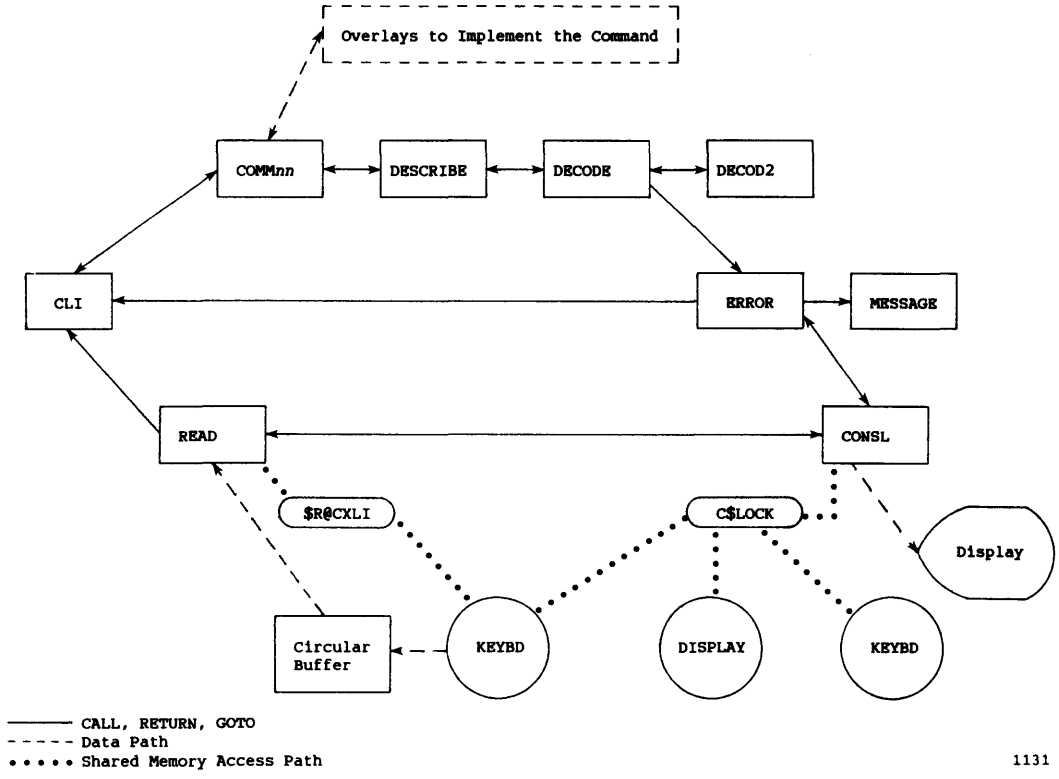


Figure 6-5. CLI Task Flow and Interaction (Does not include decimal to binary (DTB) and octal to decimal (OTB) overlays)

The flow of the STATUS command is given in figure 6-6 and the following stepflow.

Step	Function	Overlay
1.	Set the display interlock.	COMM01
2.	Save display parameters: $D@QUE Queue flags (from parameters on command) $D@FRM Display frame number (0) $D@TYP Display type (operator) $D@OVL Overlay number (DISP01) $D@PAR Display parameter (STATUS)	COMM01
3.	Activate the DISPLAY task.	COMM01
4.	Release the display interlock.	COMM01

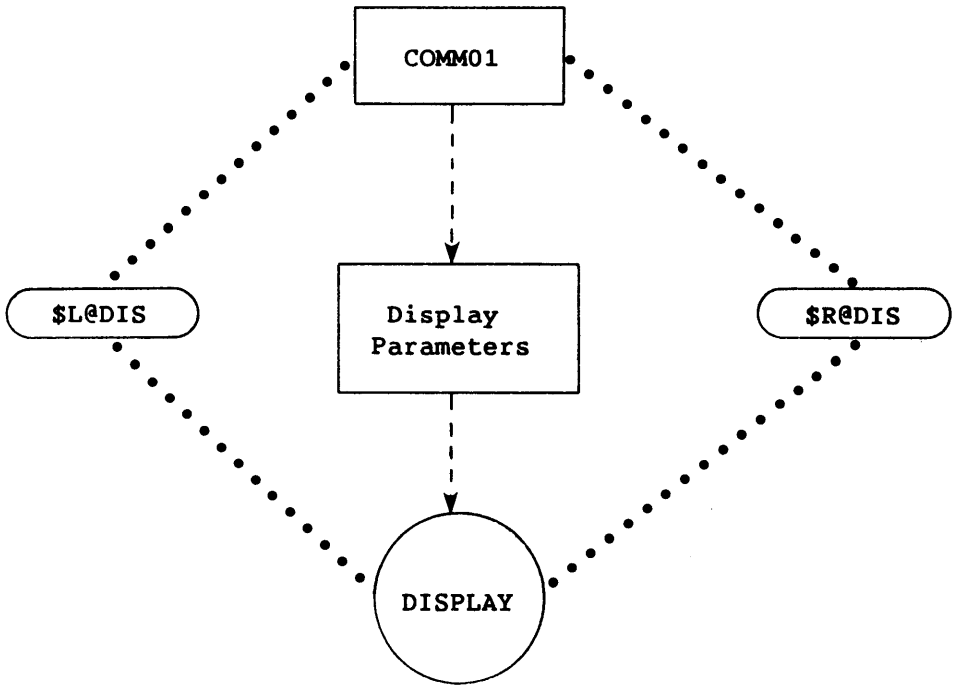


Figure 6-6. STATUS Command Flow and Interaction

The flow of the DROP command is given in figure 6-7 and the following stepflow:

Step	Function	Overlay
1.	Construct an operator function request message segment, inserting the job sequence number specified as a parameter.	COMM06
2.	If not logged on, return an error response. Send the operator function request to the mainframe and receive a response.	POST
3.	If an error response has been returned, output an error message.	COMM06 ERROR MESSAGE CONSL

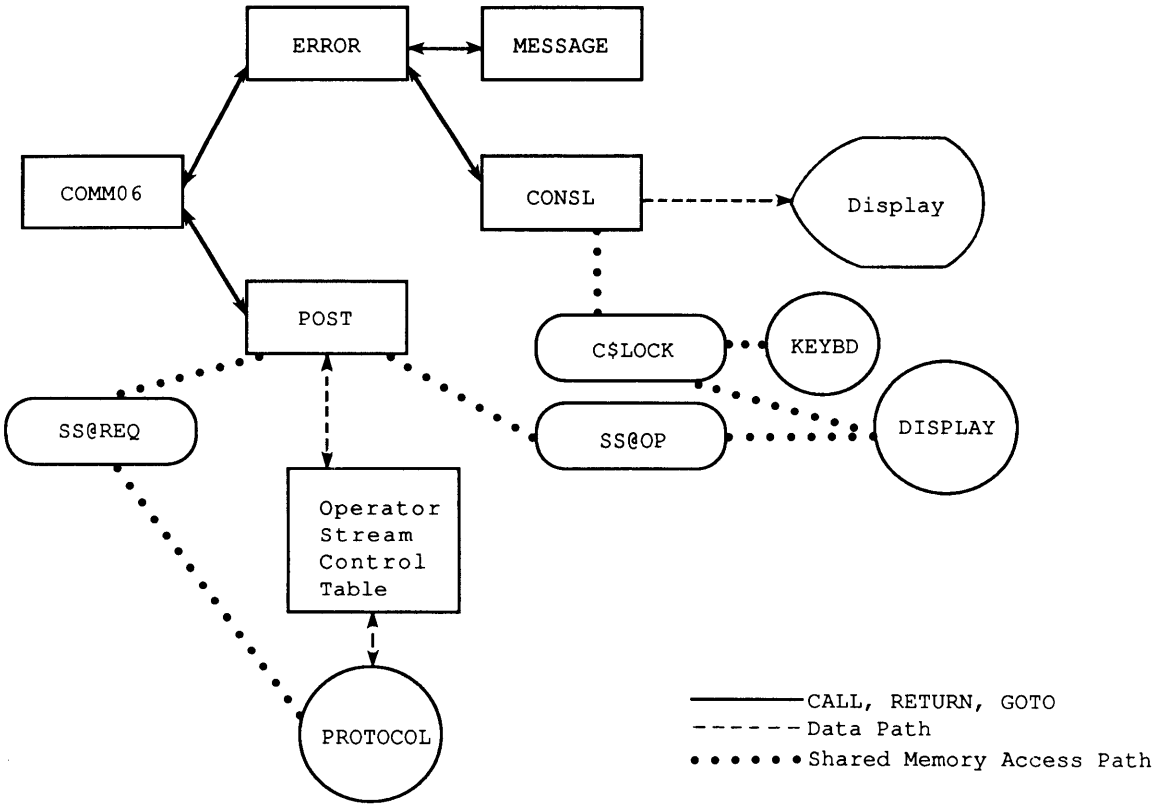


Figure 6-7. DROP Command Flow

6.3.5 PROTOCOL TASK

The PROTOCOL task controls the message interface between the IOS station and the mainframe. In general, the PROTOCOL task has the following responsibilities:

- Generates messages sent to the mainframe
- Validates the mainframe response messages
- Maintains the input and output stream states
- Creates tasks to manage input and output dataset transfers
- Monitors Operator Stream Control Table for message requests
- Monitors I/O stream control tables for message requests and stream state changes
- Schedules messages for transmission to the mainframe
- Distributes mainframe response messages

The PROTOCOL task is initiated by the LOGON command and exists as long as communications with the mainframe are maintained. Task termination may be triggered externally, by the LOGOFF command, or internally, by a breakdown in communications with the mainframe.

Table 6-5 describes the interaction areas for the PROTOCOL task.

Table 6-5. PROTOCOL Task Interaction Areas

Register	Field or Table	Use
%STAT	SS@PRO	Activation and termination communications area. The CLI, after creating the PROTOCOL task, pushes itself on the queue while awaiting a response from the PROTOCOL task initialization. For a LOGOFF command, CLI posts a termination request and discontinues its wait for an acknowledgment.
%STAT	SS@REQ	PROTOCOL task activation area. When the task is idle, it suspends by watching this area (using the WATCH macro).
%STAT	SS@ID	Station logon ID

Table 6-5. PROTOCOL Task Interaction Areas (continued)

Register	Field or Table	Use
%STAT	SS@TID	Station logon terminal identifier
%STAT	SS@POL	Interval between CONTROL messages, in tenths of a second, for an idle system
%STAT	SS@IN	Address of the first Input Stream Control Table
%STAT	SS@OUT	Address of the first Output Stream Control Table
%STAT	SS@FLG	The STA$LOG flag indicates whether the station is logged on to the mainframe. The STA$STG flag indicates whether dataset staging is enabled.
%STAT	SS@TAB	Operator Stream Control Table; used to exchange message and response information between the CLI and PROTOCOL or between DISPLAY and PROTOCOL.
%PROT	PT@TAB	Input and output stream control tables; used to exchange information between the STAGEIN and PROTOCOL tasks or between STAGEOUT and PROTOCOL tasks.
%PROT	PT@XAN	Acquire request response area. A message is posted to PT@XAN by a STAGEIN task after it processes an acquired dataset.
Queued input dataset information:		
%STAT	SS@IQ	Queueing Enabled flag
%STAT	SS@QST	Queue Status flag
%STAT	SS@QHI	High-order bits of Buffer Memory address of dataset header segment
%STAT	SS@QLO	Low-order bits of Buffer Memory address of dataset header segment
%STAT	SS@QTB	Device control table address

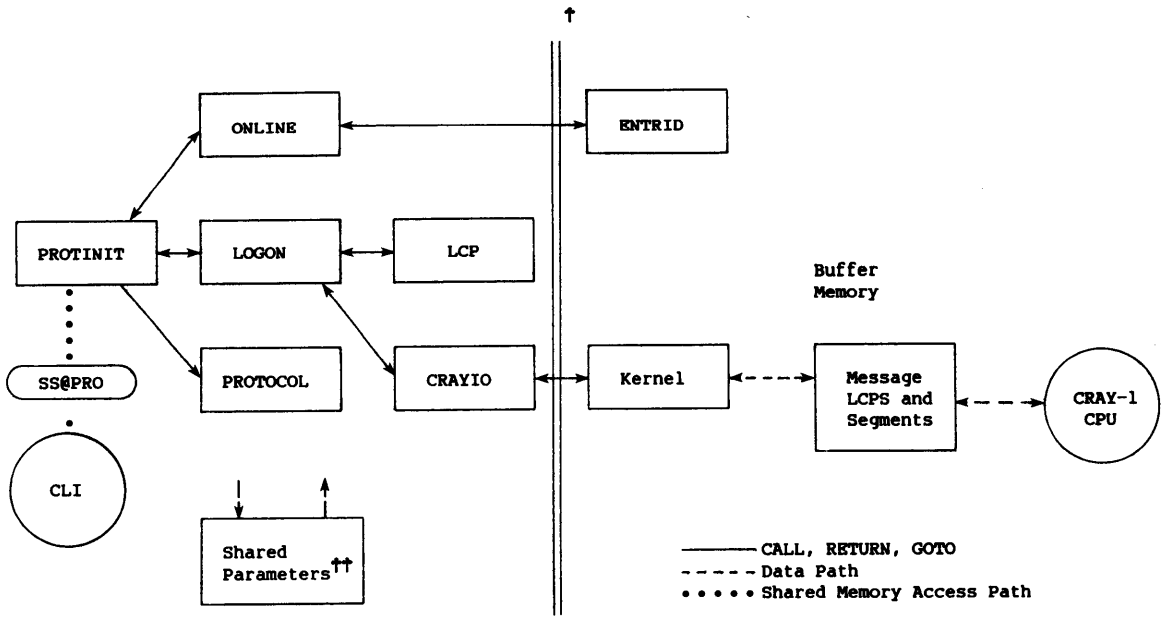
Table 6-5. PROTOCOL Task Interaction Areas (continued)

Register	Field or Table	Use
%STAT	SS@QAQ	Acquire flag
%STAT	SS@QDV	Input device
%STAT	SS@QBK	Blocking flag
%STAT	SS@QFL	Tape file number
%STAT	SS@QCC	Blocking control character

The initialization sequence for the PROTOCOL task is illustrated in figure 6-8. The steps followed in initialization are as follows:

Step	Function	Overlay
1.	Allocate and initialize Local Memory (%PROT) and the task stack area (%STACK and %LIMIT). Initialize the I/O Stream Control Tables.	PROTINIT
2.	Initialize the I/O descriptor table and the stream descriptor tables in Buffer Memory. Identify the station to the channel driver.	ONLINE ENTRID
3.	Format and send a LOGON message. Validate the START response message.	LOGON LCP CRAYIO
4.	If successful, set the Logged-on flag (STA$LOG).	LOGON
5.	Respond to the CLI task, popping it from the response queue.	PROTINIT
6.	Go to PROTOCOL.	PROTINIT

If initialization fails, all resources are released, an error response is sent to CLI, and the PROTOCOL task terminates.



† The lowest level communication routines are documented with the concentrator and are not included in subsequent diagrams.

†† Accessed by several of the overlays shown

1134

Figure 6-8. PROTOCOL Task Flow (Initialization)

The PROTOCOL task flow and interaction (for the main body of the task) are illustrated in figure 6-9. The steps followed by the main body of the PROTOCOL task are as follows:

Step	Function	Overlay
1.	Suspend waiting for a request from another task or the expiration of the poll interval.	PROTOCOL
2.	Go to PROTINIT for task termination if CLI has posted a termination request.	PROTOCOL
3.	If the Request-pending flag is set in a received LCP, return a control LCP.	PROTOCOL

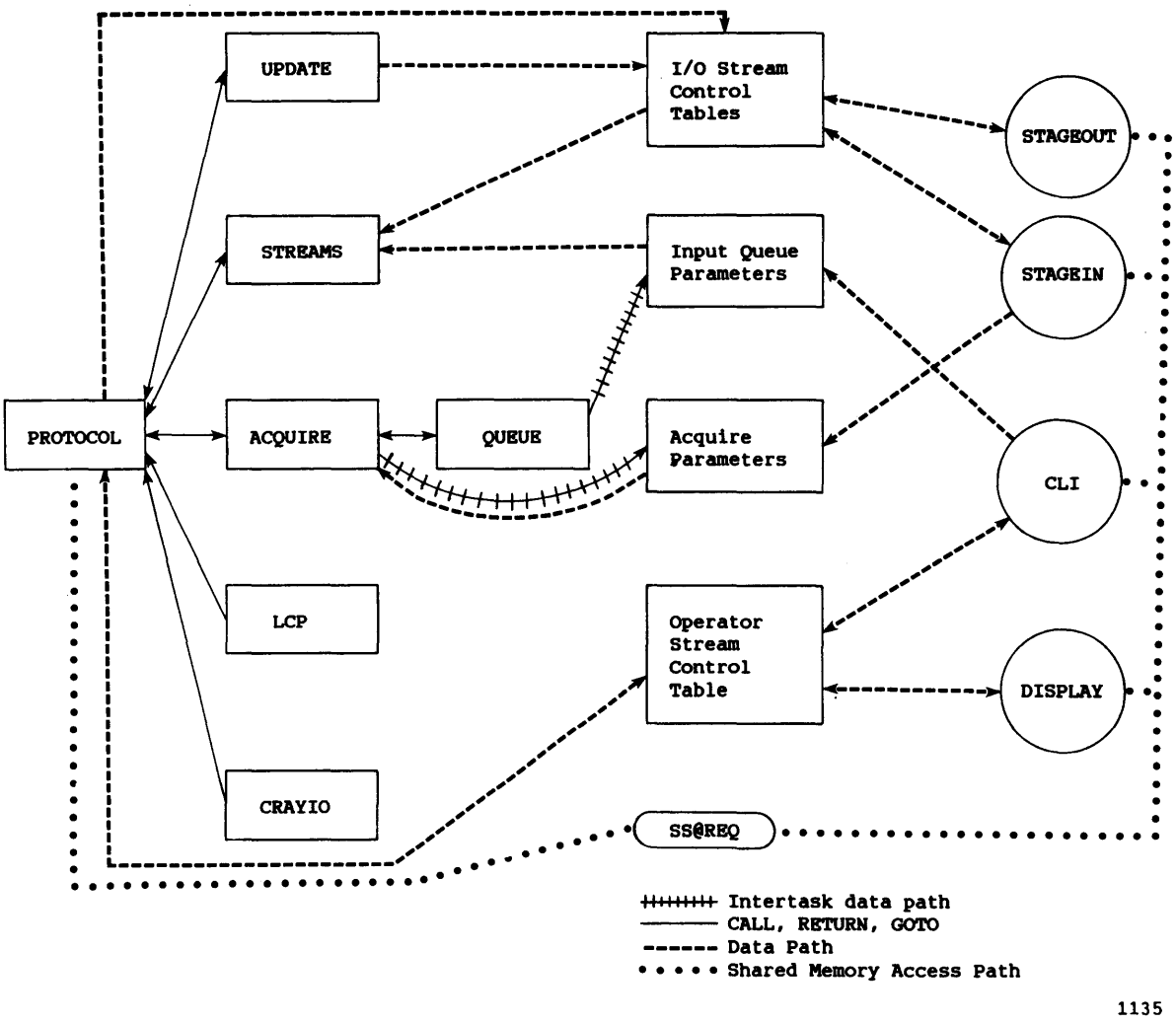


Figure 6-9. PROTOCOL Task Flow and Interaction (Main Body)

Step	Function	Overlay
4.	Schedule a dataset transfer reply message if a response has been posted (PT@PAN or PT@XAN).	ACQUIRE
5.	If a message is not scheduled and a request has been posted in the Operator Stream Control Table, schedule the operator message.	PROTOCOL

Step	Function	Overlay
6.	For each input stream, update the stream state if a request is posted in the corresponding I/O Stream Control Table. If the stream has data to send, the mainframe is ready to accept it, and no other message is scheduled, schedule the dataset header or dataset segment message for that stream.	STREAMS
7.	For each output stream, update the stream state if a request is posted in the stream control table.	STREAMS
8.	Activate an input stream if the following conditions are met: • A dataset is queued for staging. • An input stream is available. • Staging is enabled. • The input stream and total stream maximums have not been reached. If all conditions are met, a STAGEIN task is created and supplied with the input queue parameters. The stream is flagged as active, and the queue status is changed to empty.	STREAMS
9.	Activate an output stream if the following conditions are met: • The mainframe wishes to initiate staging on a stream. • Staging is enabled. • The output stream and total stream maximums have not been reached. If all conditions are met, a STAGEOUT task is created, and the stream is flagged as active.	STREAMS
10.	Clear queueing flag if all input streams are active. This prevents datasets from being queued by a SAVE or SUBMIT command or a dataset transfer request.	STREAMS
11.	Save the stream control byte (SCB) for each defined input and output stream in the LCP. The SCB is based on the stream state.	STREAMS

Step	Function	Overlay
12.	If no other message is scheduled, schedule a CONTROL message.	PROTOCOL
13.	Generate the message LCP. Send the message to the mainframe and receive the response message.	LCP CRAYIO
14.	Perform processing per the message type, one of those defined below.	PROTOCOL
For operator response message:		
15.	Validate the response message. If invalid, go to PROTINIT to handle the task termination.	LCP
16.	Save response parameters in the Operator Stream Control Table and activate the suspended task.	PROTOCOL
For dataset header or dataset segment message:		
17.	Validate the response message. If invalid, go to PROTINIT for task termination.	LCP
18.	Store response parameters in the I/O Stream Control Table and activate the STAGEOUT task.	PROTOCOL
For CONTROL message:		
19.	Validate the response message. If invalid, go to PROTINIT for task termination.	LCP
For dataset transfer request message:		
20.	Validate the response message. If invalid, go to PROTINIT for task termination.	LCP
21.	Schedule a postpone response message if another request is being processed or if the input queue is in use. Otherwise, save parameters for the input stream activation.	ACQUIRE QUEUE

For restart or message error message:

Step	Function	Overlay
22.	Go to PROTINIT for task termination.	LCP
23.	Generate a new state based on the previous state and the response SCB for each input and output stream. Store a response in the Stream Control Table and activate the corresponding STAGEIN or STAGEOUT task if so indicated by the new state.	UPDATE

For station message:

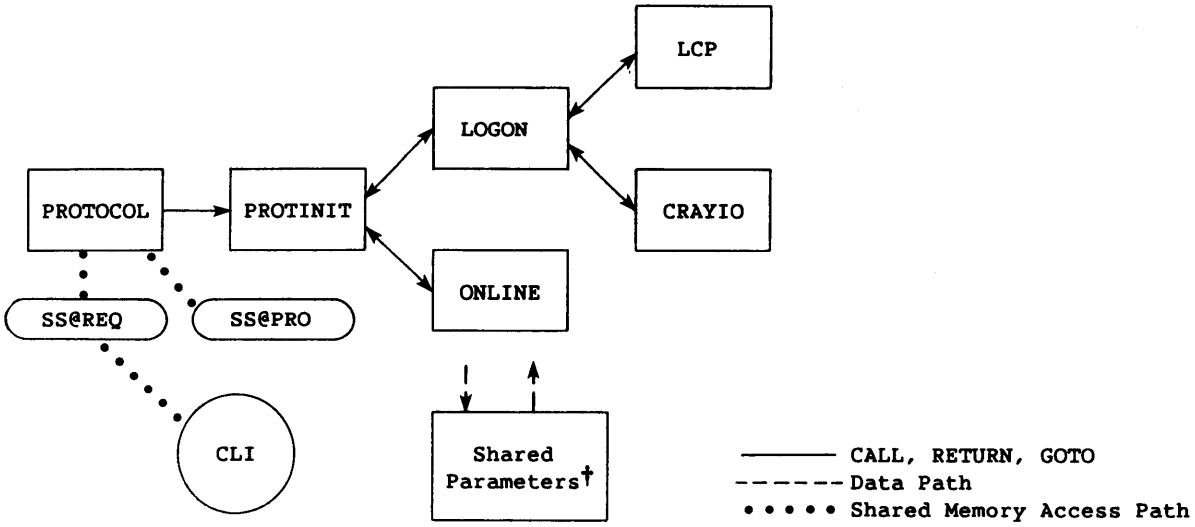
24.	When a station message is received, call overlay with pointers to the LCP and segment. If PROTOCOL gets a nonzero return from STMSG, a message response has been built. Send it immediately to SCP.	STMSG

PROTOCOL task termination is initiated by either a LOGOFF command, in which case CLI posts a message in SS@PRO, or an unrecoverable error in communications. In either case, PROTOCOL enters the overlay PROTINIT to process the task termination. Note that the overlays used for termination processing are those used for initialization. Parameters provided on the call select the desired function.

The task flow for PROTOCOL termination is illustrated in figure 6-10. The steps in termination are as follows:

Step	Function	Overlay
1.	Send a logoff message to the mainframe if termination was initiated with the LOGOFF command.	LOGON LCP CRAYIO
2.	Terminate input and output dataset transfers. Wait until all I/O Stream Control Tables are idle.	PROTINIT
3.	Disable queueing of input datasets. Release buffers reserved by queued input datasets and queued dataset transfer requests.	PROTINIT
4.	Remove station ID for the Channel Driver Table.	ONLINE

Step	Function	Overlay
5.	Release Local Memory.	PROTINIT
6.	Clear to Logged-on flag (STA$LOG).	PROTINIT
7.	Check the Operator Stream Control Table. Send an abort response if a request is outstanding.	PROTINIT
8.	Send a response to the CLI task through the communications area SS@PRO.	PROTINIT
9.	Terminate the task.	PROTINIT



† Accessed by several overlays shown

1136

Figure 6-10. PROTOCOL Task Flow (Termination)

6.3.6 STAGEIN TASK

A STAGEIN task stages a dataset from the IOS to the mainframe; one task exists for each active staging operation. The STAGEIN task is created by the PROTOCOL task when a staging request is received. The request may originate from a SAVE or SUBMIT command or from an ACQUIRE or FETCH message from the mainframe. Table 6-6 describes the task interaction areas for STAGEIN.

Table 6-6. STAGEIN Task Interaction Areas

Register	Field or Table	Use
--	Stream Control Table	Exchanges protocol message request and response information between the STAGEIN and PROTOCOL tasks. The PROTOCOL task passes the address as a parameter when it creates the STAGEIN task.
%STAT	SS@REQ	PROTOCOL task activation area
%PROT	PT@XAN	Response word for acquire requests. The address of this word is a parameter supplied by the PROTOCOL task at STAGEIN creation.

The task flow for STAGEIN is given in figure 6-11 and in the following stepflow:

Step	Function	Overlay
1.	Open the stream and send the dataset header to the mainframe. The Buffer Memory address of the dataset header is one of the parameters supplied by PROTOCOL at creation.	STAGEIN STIO
2.	Initialize the Stream Control Table, then go to the overlay controlling input staging; the name of the overlay is dictated by the acquisition code: MT STTAPI Expander tape ST STXDKI Expander disk	STAGEIN

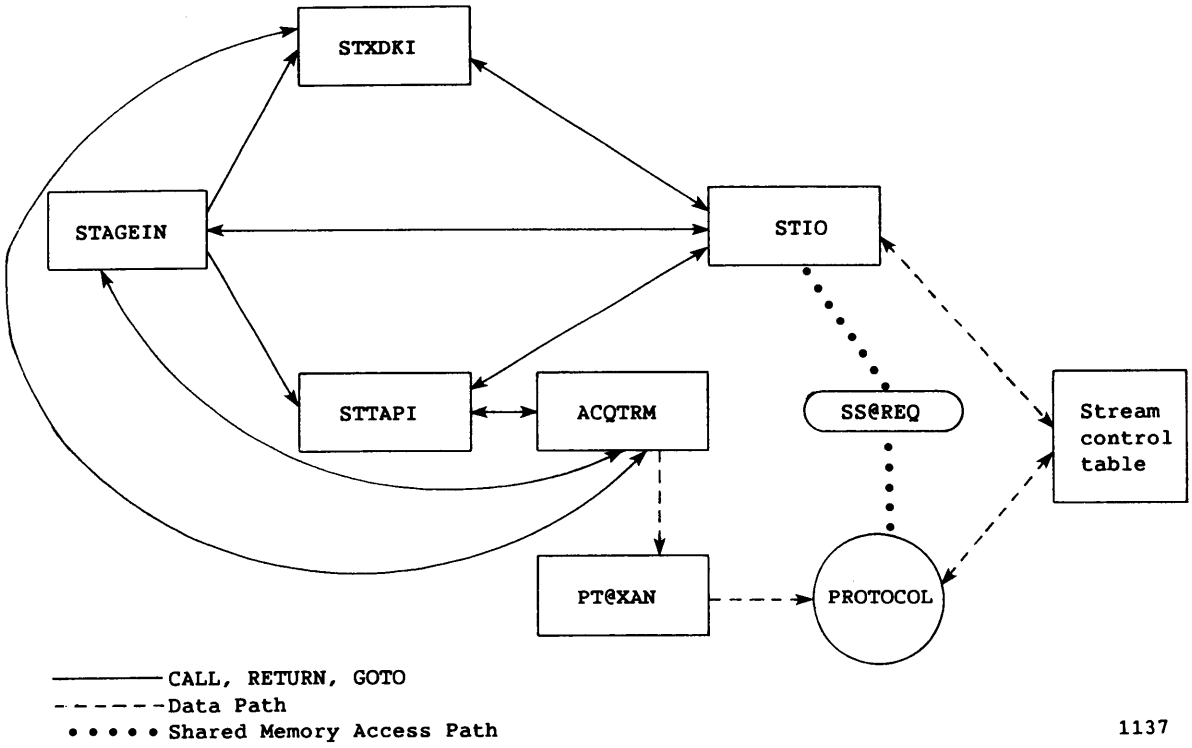


Figure 6-11. STAGEIN Task Flow and Interaction

For tape input, the STAGEIN task opens the device, which generates a tape mount request on the MIOP Kernel console. The STAGEIN task reads a block of data from tape into Local Memory, writes the data to a buffer in Buffer Memory, and sends it to the mainframe.

For disk input, the STAGEIN task opens the device, which generates a disk mount request on the MIOP Kernel console if the requested volume is not mounted. The STAGEIN task then reads a block of data from disk into Local Memory, writes the data to a buffer in Buffer Memory, and sends it to the mainframe.

When the end of the input file is encountered or if an error occurs during the staging process (for instance, an abort response from the PROTOCOL task or an error on a read operation), all resources allocated by the task must be released. Any or all of the following operations may be required.

- Release Local Memory buffer
- Release Buffer Memory buffer
- Post a response to the acquire request

- Free the input stream
- Release the Stream Control Table
- Terminate the STAGEIN task

6.3.7 STAGEOUT TASK

The STAGEOUT task stages a dataset from the mainframe to the IOS; one task exists for each active staging operation. The STAGEOUT task is created by the PROTOCOL task when the mainframe initiates staging on an output stream. Table 6-7 describes the task interaction areas of STAGEOUT.

Table 6-7. STAGEOUT Task Interaction Areas

Register	Field or Table	Use
--	Stream Control Table	Used for STAGEOUT-PROTOCOL task information interchange. The PROTOCOL task passes the address as a parameter when it creates the STAGEOUT task.
%STAT	SS@REQ	PROTOCOL task activation area

The task flow and interaction for STAGEOUT is shown in figure 6-12. The steps in the task flow are as follows:

Step	Function	Overlay
1.	Open the stream and read the dataset header.	STAGEOUT
2.	Initialize the Stream Control Table.	STAGEOUT
3.	Go to the overlay controlling output staging; the name of the overlay is dictated by the disposition code:	STAGEOUT

MT	STTAPO	Expander tape
PR	STUBPR	Expander printer
PT	STPLOT	Expander plotter
ST	STXDKO	Expander disk

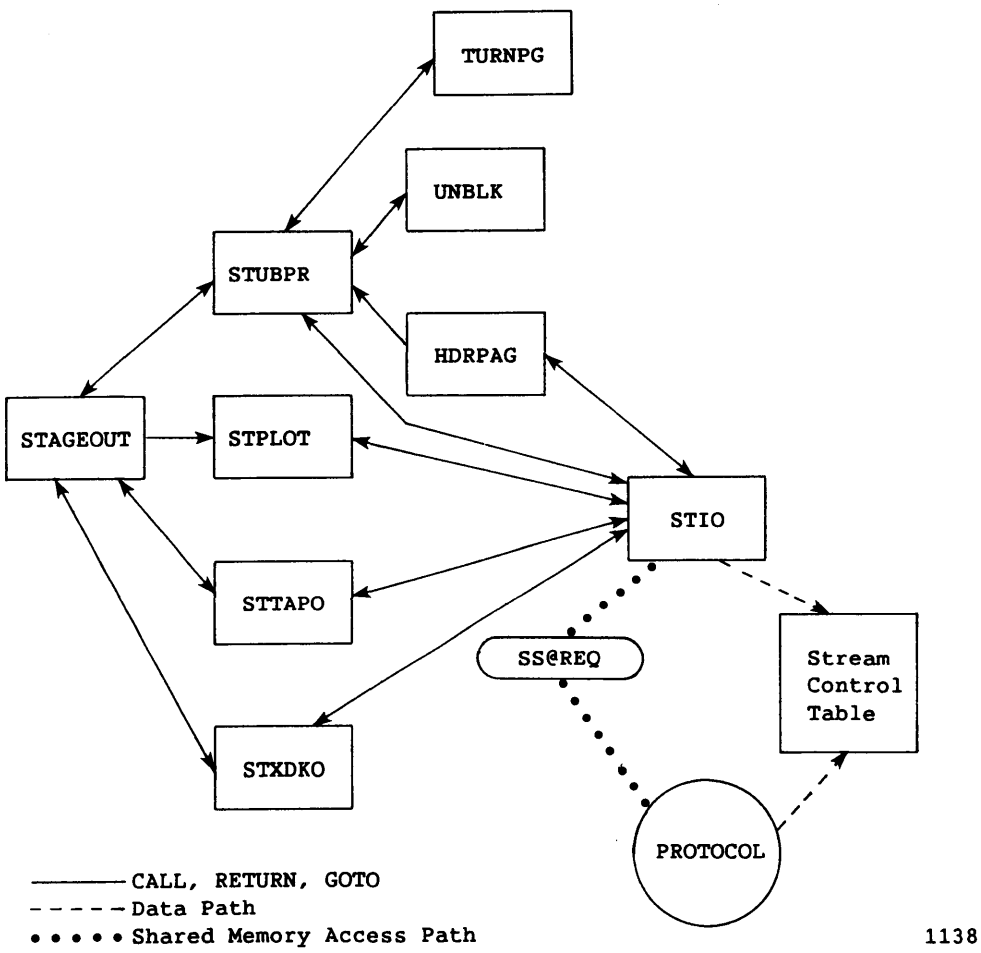


Figure 6-12. STAGEOUT Task Flow and Interaction

For tape output, the STAGEOUT task opens the device, which generates a tape mount request on the MIOP Kernel console. The STAGEOUT task then receives a segment of data, copies the data into Local Memory from Buffer Memory, releases the Buffer Memory block, and writes the data to tape.

For disk output, the STAGEOUT task opens the device, which generates a disk mount request on the MIOP Kernel console if the requested volume is not mounted. The STAGEOUT task then receives a segment of data, copies the data into Local Memory from Buffer Memory, releases the Buffer Memory block, and writes the data to disk.

For printed output, STAGEOUT generates a header page for the listing. It prints each segment of data received from the mainframe. Each line contained in the segment is copied to Local Memory, deblocked, and printed before the Buffer Memory block is released. If data is to be

printed in document format, each line is written to Buffer Memory instead of to the printer. When a full page has been gathered, the TURNPG routine is called to format and print the page.

Plot output is handled like the printed output except that a header page is not produced and deblocking of the output is not required.

At the conclusion of the transfer or if an error is encountered (for instance, an abort response from PROTOCOL or an error during a write operation), some or all of the following functions must be performed:

- Release Local Memory buffer
- Release Buffer Memory buffer
- Close the device
- Free the stream
- Release the Stream Control Table
- Terminate the STAGEOUT task

6.3.8 STIO OVERLAY

The STAGEIN and STAGEOUT tasks communicate with the PROTOCOL task using the I/O Stream Control Tables (SCTs). An I/O task stores stream state and message control parameters in its SCT. The PROTOCOL task uses this information to generate messages for the mainframe and to validate response messages. When appropriate, it stores response parameters in the SCT and reactivates the controlling I/O task. The I/O tasks use the STIO overlay to interface with the PROTOCOL task. For a STAGEIN task, STIO is activated with a CALL service request of the following format:

Location	Result	Operand
	CALL	STIO,(*fcode,table,sgn,sgbc,segu,segl*)

fcode Function code, as follows:

ISF$WRIT	Send dataset header message to mainframe if *sgn*=0. Send dataset segment message to mainframe if *sgn*≠0.
ISF$END	Send END SCB, release SCT, and terminate task.
ISF$CAN	Send CAN SCB, release SCT, and terminate task.
ISF$PPN	Send PPN SCB, release SCT, and terminate task.
ISF$DONE	Release SCT and terminate task.

table SCT address

sgn Stream segment number

sgbc Segment bit count

segu High-order bits of Buffer Memory address of message segment

segl Low-order bits of Buffer Memory address of message segment

sgn, *sgbc*, *segu*, and *segl* are meaningful only if *fcode* is ISF$WRIT. If the function code is ISF$WRIT, STIO stores *sgn*, *sgbc*, *segu*, and *segl* in the appropriate SCT fields. It also stores a message descriptor in the SCT (dataset header message descriptor if *sgn*=0; otherwise, dataset segment message descriptor). STIO then stores a function code in the SCT, activates the PROTOCOL task if necessary, and stops, waiting for a response. Because of the asynchronous nature of the tasks and the fact that state changes may be initiated by the mainframe (for instance, postponing the transfer) or the PROTOCOL task (for instance, postponing because of a logoff), the SCT may already contain a response.

The PROTOCOL task, which monitors the SCTs for requests, ultimately schedules and sends the requested message to the mainframe. A response code is stored in the SCT, and the STAGEIN task is reactivated. The STIO overlay returns the response to the calling overlay in the A register. The responses and their significance are as follows:

Response	Meaning
ISR$OK	OK; proceed with next request.
ISR$PPN	Postpone the dataset transfer; the next function must be ISF$DONE.
ISR$CAN	Cancel the dataset transfer; the next function must be ISF$DONE.

For function codes ISFEND, ISFCAN, and ISF$PPN, STIO stores only the function code in the SCT for processing by the PROTOCOL task. These functions cause PROTOCOL to send the appropriate SCB to the mainframe. When a response is received, the SCT is released and the task terminated. No response is possible; the calling overlay is required to release all resources before issuing these functions.

Function ISF$DONE causes STIO to release the SCT and terminate the task. No communication with the PROTOCOL task is performed.

For a STAGEOUT task, a call to STIO takes the following form:

Location	Result	Operand
	CALL	STIO,(*fcode*,*table*,*sgn*,RO=*sgbc*,RO=*segu*, RO=*segl*)

fcode Function code, as follows:

OSF$READ	Read dataset header (*sgn*=0).
OSF$SVD	Send SVD SCB, release SCT, and terminate task.
OSF$PPN	Send PPN SCB, release SCT, and terminate task.
OSF$CAN	Send CAN SCB, release SCT, and terminate task.
OSF$DONE	Release SCT and terminate task.

table SCT address

sgn Expected segment number

RO=*sgbc* Register in which segment bit count is returned

RO=*segu* Register in which the high-order bits of the Buffer Memory address of the message segment are returned

RO=*segl* Register in which the low-order bits of the Buffer Memory address of the message segment are returned

Parameters *sgn*, *sgbc*, *segu*, and *segl* are meaningful only if the function code is OSF$READ. If the function code is OSF$READ, STIO stores a message descriptor in the SCT (dataset header descriptor if *sgn*=0; otherwise, dataset segment descriptor), posts a read function code, activates the PROTOCOL task, and waits for a response.

The PROTOCOL task informs the mainframe that data can be sent on the output stream (an RCV SCB is posted). When the mainframe responds with data for the stream, the PROTOCOL task validates the message using the descriptor in the SCT. The PROTOCOL task stores the message segment bit count, segment address, and a response code in the SCT and reactivates the STAGEOUT task. STIO loads the returned parameters from the SCT and returns them to the calling overlay. The response code, which is returned in the A register, may be one of the following:

Response	Meaning
OSR$OK	OK; proceed with the next request.
OSR$END	END SCB received; the next function must be OSFSVD, OSFPPN, or OSF$CAN.
OSR$CAN	CAN SCB received; the next function must be OSF$DONE.

Functions OSFSVD, OSFPPN, and OSF$CAN send a function to the PROTOCOL task. After the function is performed, the task is terminated. An OSF$DONE function terminates the task immediately.

6.3.9 POST OVERLAY

The CLI and DISPLAY tasks communicate with the PROTOCOL task through the Operator Stream Control Table. The interlock SS@OP is associated with the table to prevent both CLI and DISPLAY from accessing the table simultaneously. The CALL macro for the POST overlay takes the following form:

Location	Result	Operand
	CALL	POST,(*glcp*,*segment*,*sgbc*,*vlcp*,*flags*, RO=*upper*,RO=*lower*,RO=*rbc*)

glcp	Message descriptor controlling the LCP format
segment	Local Memory address of message segment
sgbc	Segment bit count
vlcp	Response message descriptor
flags	POS$NO; release response message segment.
RO=*upper*	Register to receive the high-order bits of the Buffer Memory address of the response message segment
RO=*lower*	Register to receive the low-order bits of the Buffer Memory address of the response message segment
RO=*rbc*	Register to receive the response message segment bit count

If *flags*=POS$NO, *upper*, *lower*, and *rbc* are used.

The POST overlay stores the output message parameters (*glcp*, the address of the segment written to Buffer Memory, *sgbc*, and *vlcp*), sets a flag to request processing, activates the PROTOCOL task, and suspends itself.

The PROTOCOL task ultimately schedules the message for output; it uses the message descriptor to generate the LCP. When a response message is received, the PROTOCOL task validates it using the response message

descriptor. It stores the segment address and bit count in the Stream Control Table, activates the CLI or DISPLAY task, and returns a response code to that task.

The POST routine loads parameters from the Operator Stream Control Table and releases the segment or saves the parameters, depending on the *flags* specification. It returns to the calling overlay, returning the response code in the A register.

6.4 GLOBAL SYMBOLS

A partial list of the types of global symbols is as follows:

Type	Definition and Use
Function code	An overlay may provide more than one function. A function code, passed as a parameter, selects the function to be performed. For example, CLINIT performs both initialization and termination processing for the CLI task. The function is selected by the function code CLI$INIT or CLI$TERM.
Error code	An error code is generated by the ERCODE macro. The error code includes overlay number and message offset information that allows the MESSAGE overlay to access message text and formatting data. Error codes are defined in the SYSTEXT overlay.
LCP descriptors	LCP descriptors are pointers to LCP Descriptor Tables in the LCP overlay. The descriptors are generated by the LCP macro. The LCP overlay uses the descriptors to construct or validate message LCPs.
Display parameters	The display parameters, DP$*disp* and DV$*disp*, identify the display type and the overlay that initiates the display. The parameters are defined by the DISPARS macro in the OVLNUM overlay.
Command parameters	The command parameters, CP$*comm* and CV$*comm*, identify the command type and the overlay that processes the command. The parameters are defined by the COMPARS macro in the OVLNUM overlay.

Type	Definition and Use
Stream states	The input and output stream states are defined in the UPDATE overlay by the STATE macro. The symbol defines the stream state and includes the following information: • SCB to send in next message • Stream available for assignment • Stream ready to send or receive data • State transition table address The state transition table address is an offset into the UPDATE overlay. It points to a table used to validate the SCB in the response LCP and identify the succeeding stream state.
Parameter descriptors	The parameter descriptors are offsets into the SYNTAX overlay that are generated by the PARAMS macro. The descriptors reference a table describing the parameters associated with a particular command.

6.5 CONSOLE OUTPUT

The CONSL overlay is called to handle all console output and special functions (for instance, ring console bell or scroll the screen). The output is not formatted for a particular device type; it must be translated to the codes and control characters used for a specific console. Currently, the station supports the AMPEX Dialogue 80, the SOROC IQ-120, and the TEC 455 consoles. The AMPEX, SOROC, and TEC455 overlays handle the output translation for the respective devices.

6.6 SCREEN IMAGE

The station maintains an image of the console screen in Buffer Memory. Output destined for a particular location on the screen is compared to the existing data and, in general, is not output if it is unchanged. Pressing the ESCAPE key at the station causes the entire screen to be rewritten from the image buffer. The image buffer is also used for the SNAP command, which prints an image of the station screen.

7. FRONT-END CONCENTRATOR

The I/O Subsystem (IOS) concentrator relieves the mainframe from the burden of handling the interrupts for each subsegment of messages transferred between the mainframe and attached front-ends. The concentrator looks exactly like a Cray channel pair to a front-end, so no changes are necessary in existing front-end stations. The concentrator can handle data from multiple IDs through one channel. Any front-end ID may send subsegments of variable lengths to the concentrator. All segment buffers are allocated dynamically. (This connection is used only by the COS operating system and supports only the Station Call Processor (SCP) protocol.) Refer to section 11, Front-End Interface Logical Path Activity, for information on the UNICOS operating system.

The concentrator software will support the receiving of additional input data, from the station during subsegment transfers, beyond what was expected. This allows stations, whose channel width is not the same as the IOP channel width, to use segment sizes that are not multiples of the two channel widths. This feature is controlled with the $APTEXT parameter I@XTRA. The value of I@XTRA is the number of additional parcels of input data that may be received beyond the segment size and not be considered an error. Only valid data in the segment will be transferred to the mainframe.

Figure 7-1 shows the structure of the concentrator software.

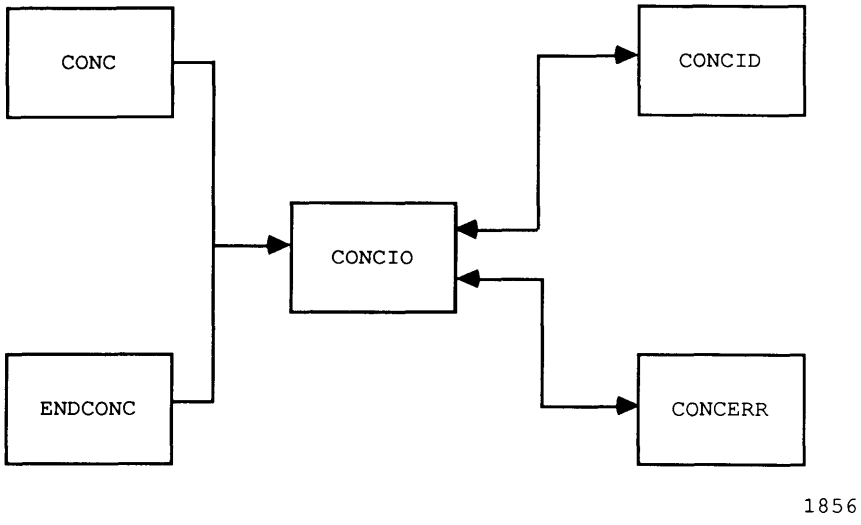


Figure 7-1. Tree Structure of Concentrator Software

7.1 CONC OVERLAY DESCRIPTION (CONCENTRATOR INITIALIZATION)

Overlay CONC is activated by the Kernel command CONC or by a master operator entering a CHANNEL ON command that specifies an I/O Processor (IOP) concentrator. The Kernel console output contains descriptive messages that indicate successful or unsuccessful completion of the concentrator initialization. A flow description follows:

```
Call CRAY overlay to ensure IOP/Cray are linked
Determine the channel ordinal of requested concentrator
Locate the Front-end Interface (FEI) table entry for this ordinal
Validate the requested ordinal.  For example, check the channel type,
  current status.
Clear the IOP channel pair
Set hold disconnect on the output channel
Get Local Memory for the concentrator table
Empty the input channel of any residual data
Do a Port Select function is it is a VAX interface
Set flags indicating that this FEI is initialized and active
Create CONCIO activity
Send console message detailing the fate of the initialization
  procedure
Terminate
```

7.2 CONCIO ACTIVITY DESCRIPTION

Upon entry, the CONCIO global register (LOCAL) points to the Local Memory table for this concentrator. Registers LCP0 and LCP1 point to the LCP Buffer Memory address. Register CHAN contains the associated input channel number. Register MCO is the channel ordinal, and register FEI points to the proper FEI table entry.

Physical I/O requests are asynchronous and are executed in the overlay code itself. The activity does not surrender control when initiating I/O. Upon an interrupt, the Kernel SIGNALS an I/O queue determined from the associated FEI table entry. A flow description follows:

```
Get addresses of input and output LCPs from the concentrator
  table (CT@ILC, CT@OLC).  Also get concentrator's address
  of the I/O queue (CT@IOQ).
Initialize flags.
Send a restart LCP to the front-end station (FE).

RESTRT program label
     Open the input channel for a logon-LCP (6 words) from the front
       end.
```

```
WATCH program label
     Store "waiting for input" in the concentrator's status field
       (CT@ST).
     UNTIL "input" is signaled on the concentrator's I/O queue (IOQ):
          WATCH the IOQ; time-out value equals approximately three
          seconds.
          Get the value of the front end interface termination flag
                 (FEI@TM).
          IF FEI@TM is nonzero, GOTO program label TERMIN.
     ENDTIL

INPUT program label
   Store "inputting" in the concentrator table's status field (CT@ST).
   RETURN-JUMP to program label CHKIO.
   IF CHKIO returned an error status, GOTO program label ERROR.
   CALL CONCID to find the entry in the concentrator ID table.
   IF CONCID returns an error code, GOTO program label ERROR.
   Get the Channel Extension Table(CXT) address from the
     concentrator ID table (CE@CXT).
   IF use of link trailer packets (LTPs) is flagged for this FE
    (CE@LTP), THEN get the address of where to put LTPs for this
    concentrator (CT@LTP).
   Get the variable subsegment sizes flag (CE@VSS).
   IF the station accepts variable subsegment sizes THEN
      IF this is a logon-LCP THEN
        Set subsegment size to 6 words.
      ELSE
        get the size out of the LCP's segment bit count (LC@BCU &
         LC@BCL).
      ENDIF
   ELSE
      get the size from the concentrator ID table (CE@SGZ).
   ENDIF
   Get the number of subsegments out of the input-LCP (LC@NSS).
   IF there is a data segment or an LTP to follow this input LCP
       (ILCPP),THEN
      Using the number of subsegments, the LTP flag (1 or 0), the
      subsegment size, and the maximum amount of extra input likely
      to be received, allocate the appropriate number of I/O buffers
      (maximum of 2).  Determine the amount of the next transfer by
      executing a RETURN-JUMP to program label SETXLEN.  Start the
      read from the station.
   ENDIF
   Get the Cray's ILCP address from the CXT table (CXCIL0 & CXCIL1).
   IF there is no ILCP address or this is a logon segment, THEN
      Poll the Cray for the ILCP address.
      IF a poll error occurred, THEN
        GOTO program label ERROR.
      ELSE
        Copy the DAL to the CXT and release the DAL.
```

```
   ENDIF
   Clear the logon and type flags of the CXT (CXLOG &  CXTYPE).
   Get the new Cray ILCP address from the CXT (CXCIL0 & CXCIL1).
ENDIF
Write the ILCP to the Cray on the high-speed channel.
Get the Cray's input segment address from the CXT (CXISG0 &
  CXISG1).
UNTIL this segment is all read in from the station :
   WATCH the IOQ for a completion signal from the previous I/O;
     this watch times-out after ten seconds.
   IF the channel times-out, GOTO program label ERROR.
   RETURN JUMP to program label CHKIO.
   IF CHKIO returns an error status, GOTO program label ERROR.
   IF this is a logon segment, THEN
     Get the subsegment size from the logon message segment
     (LM@SSG), convert to parcel size,& store in the concentrator
     ID table (CE@SGZ).  Get the variable subsegment size flag out
     of the logon message segment (LM@VSS) and store in the conc.
     ID table (CE@VSS).  If the checksum enabled flag in the logon
     message segment (LM@CKZ) is set, THEN set the LTP flag in the
     concentrator ID table (CE@LTP) and get from the concentrator
     table the address in which to read LTPs (CT@LTP).
   ENDIF
   Switch to the local memory buffer least recently used.
   IF there are more subsegments to transfer, THEN
     IF the current subsegment has been completely transferred,
       THEN Decrement the number of segments left to transfer.
     ENDIF
   ELSE
     Get the Cray LTP address out of the CXT (CXILT0 & CXILT1).
   ENDIF
   IF there is more data or an LTP to be transferred, THEN
     Determine the next I/O length.
     Input the determined amount from the FE.
     Write the information to the Cray over the high-speed channel.
   ENDIF
     IF there is more data to transfer, THEN increment the Cray
       segment address by the length of the last transfer.
ENDTIL
Store "waiting for output" in the concentrator table's status
  field (CT@ST).
Poll the Cray for a new CXT.
IF a poll error, THEN
   GOTO program label ERROR.
ELSE
   Copy the DAL to the CXT and release the DAL.
ENDIF
Get the number of subsegments out of the CXT (CXTNSS).
IF the variable subsegment size flag is set, THEN
   IF the number of subsegments is nonzero, set it to one.
   Get the Cray subsegment size from the CXT (CXLSEG)& convert to
     parcels.
```

```
ENDIF
Using the number of subsegments and the subsegment size, allocate
  the appropriate number of I/O buffers (maximum of two).
IF an LTP is expected by the FE, THEN get the concentrator table's
  address of where to put the LTP (CT@LTP).
Store "outputting" in the concentrator table's status field
  (CT@ST).
Get the Cray's output-LCP (OLCP) address from the CXT (CXCOL0 &
  CXCOL1).
Read the OLCP from the Cray on the high-speed channel.
Open the I/O channel for an ILCP from the FE.
Send the Cray's OLCP to the FE.
Get the Cray's output segment address from the CXT (CXOSG0 &
  CXOSG1).
UNTIL there is no more data to write to the FE :
   IF there is more subsegment data or an LTP, THEN
     IF there is subsegment data, THEN
       Determine the next I/O length.
       Set flag to send a channel disconnect if this transfer is
         the last for this subsegment.
     ELSE
       Get the Cray's address of the LTP from the CXT (CXOLT0 &
         CXOLT1).
       Set flag to send a channel disconnect.
     ENDIF
       Read the segment data/LTP from the Cray on the high-speed
         channel.
       Flag that there is data ready to be written to the FE.
       Increment the Cray's segment address pointer by the length
         of the last transfer.
   ENDIF
   WATCH the IOQ for completion of the last I/O to the FE; this
   WATCH times-out after three seconds.
   IF the WATCH timed-out, THEN
     GOTO program label ERROR.
   ELSEIF the previous I/O returned an error status, THEN
     GOTO program label ERROR.
   ELSEIF the WATCH returned a status indicating that the
     concentrator has been sent input from the FE, THEN
     Prepare to verify the input interrupt.
     GOTO program label INPUT.
   ELSEIF the WATCH returned a status indicating both output and
     input interrupts have arrived, THEN
     Save output channel's address and transfer length; re-open
     the input channel; restore output channel's parameter's so
     that the output transfer may be verified.
   ENDIF
   GOTO program label CHKIO.
   IF CHKIO returned an error status, go to program label ERROR.
   IF have just finished transferring a complete subsegment and
     there are more subsegments to transfer, decrement the
     subsegment count.
```

```
      IF there is data ready to be written to the FE, THEN
        Clear the data ready flag.
        Write the data to the FE.
      ENDIF
   ENDTIL
   Get the acknowledgment flag out of the CXT (CXACK).
   IF the acknowledgment flag is set, THEN
      Poll the Cray with the done flag set in the CXT & without
        activating the SCP task in COS.
      IF there was a poll error, THEN
        GOTO program label ERROR.
      ELSE
        Copy the DAL to the CXT and release the DAL.
      ENDIF
   ENDIF

TERMIN program label
   Release any I/O buffers that were acquired.
   If this ID is being terminated, THEN
      CALL CONCID to terminate this ID's concentrator ID table entry.
      Release the concentrator table's memory.
      Clear the table address, terminating flag, and active flag in
        the FEI table (FEI@TB, FEI@TM, FEI@AC).
   ENDIF
   Prepare to read in another ILCP.
   GOTO program label WATCH.

ISSUE program label
   Store the "hold channel disconnect" flag for the KERNEL (CT@HLD).
   Function the channel for the transfer (input or output) to the FE.

CHKIO program label
   IF there is no I/O error currently detected, THEN
      Compare the actual channel address to that expected following
        the last transfer.
      IF there is a mismatch in addresses, THEN set the error code.
   ENDIF

DISC program label
   Send a single disconnect on the output channel.

ERROR  program label
   Get the current concentrator state out of the conc. table (CT@ST).
   CALL CONCERR to display the error message.
   IF the current concentrator state equals "outputting", and no poll
      error is present, THEN
      RETURN jump to program label DISC.
   GOTO program label RESTRT.
```

7.3 CONCID OVERLAY DESCRIPTION

The CONCIO activity calls overlay CONCID to locate ID-based table entries. It links new entries, unlinks logged-off entries, and issues LOGOFF LCP requests to the Cray mainframe upon concentrator termination. The Kernel console receives descriptive messages on a LOGON/LOGOFF request and a concentrator termination. A flow description follows:

```
Initialize registers
IF registering/deregistering an ID, THEN
   UNTIL all ID entries have been searched, or the selected entry has
   been found ...
      Get current entry pointer
      IF Log Off Requested flag set, THEN
        Unchain this entry from the ID entry chain
        Release this entry's memory space
      ENDIF
   ENDTIL
   IF a LOGON request, THEN
      IF the entry table is not found, THEN
        Get memory for a new entry table
        Store ID in the new entry table
        Link new entry to chain of entries
      ELSE
        Clear flags in the entry table
      ENDIF
      Store logon segment size in entry
      Flag concentrator activity that this is a logon request
      Initialize the CXT
   ELSE
      Set logged-off flag in entry table
   ENDIF
ELSE
   Send LOGOFF LCPs to mainframe for each entry
   Unqueue entry table and release memory
   Send logged-off message for each ID to the Kernel console
ENDIF
Send Kernel console message
Retrun entry address and logon-request flag if appropriate
```

7.4 CONCERR OVERLAY DESCRIPTION

The CONCERR overlay handles concentrator errors. A flow description follows:

```
Empty and clear the IOP channel pair if non-NSC channels
Issue a Port-Select function if this is a VAX interface
Issue a descriptive error message to the Kernel console
Return
```

7.5 ENDCONC OVERLAY DESCRIPTION

The ENDCONC overlay initiates concentrator termination. The Kernel console receives a descriptive message if ENDCONC cannot process the termination request.

```
Determine the specified concentrator channel ordinal
Locate the FEI table entry for this channel ordinal
Validate the specified channel ordinal
Set the Termination flag in the FEI entry
Terminate
```

8. INTERACTIVE STATION

The interactive station is a set of tasks running in the Master I/O Processor (MIOP) that permits consoles connected directly to the MIOP to become attached to mainframe jobs. A job is created in the mainframe when an interactive console logs on.

This station is composed of two parts: the interactive concentrator and the interactive console. The interactive concentrator gathers messages from the consoles, sends them to the mainframe, receives responses, and distributes them to the console routines. The interactive console routines handle the input and output to and from the consoles and prepare messages to be sent to the mainframe through the interactive concentrator.

The structure of the interactive concentrator is illustrated in figure 8-1. Figure 8-2 defines the structure of the interactive console routines.

The commands described in this section must be followed by a carriage return.

8.1 INTERACTIVE CONCENTRATOR OVERLAYS

The overlays described in the following subsections constitute the interactive concentrator.

8.1.1 IAIOP OVERLAY

The IAIOP overlay initializes the interactive concentrator and processes commands for it. IAIOP creates the task IAIOP1, which is the control overlay for the interactive concentrator.

The interactive concentrator supports the following commands: LOG, LOGOFF, POLL, and END. Each command is preceded by IAIOP.

The LOG command logs on the interactive concentrator and initializes it if it is not already initialized. This command is entered at the Kernel console.

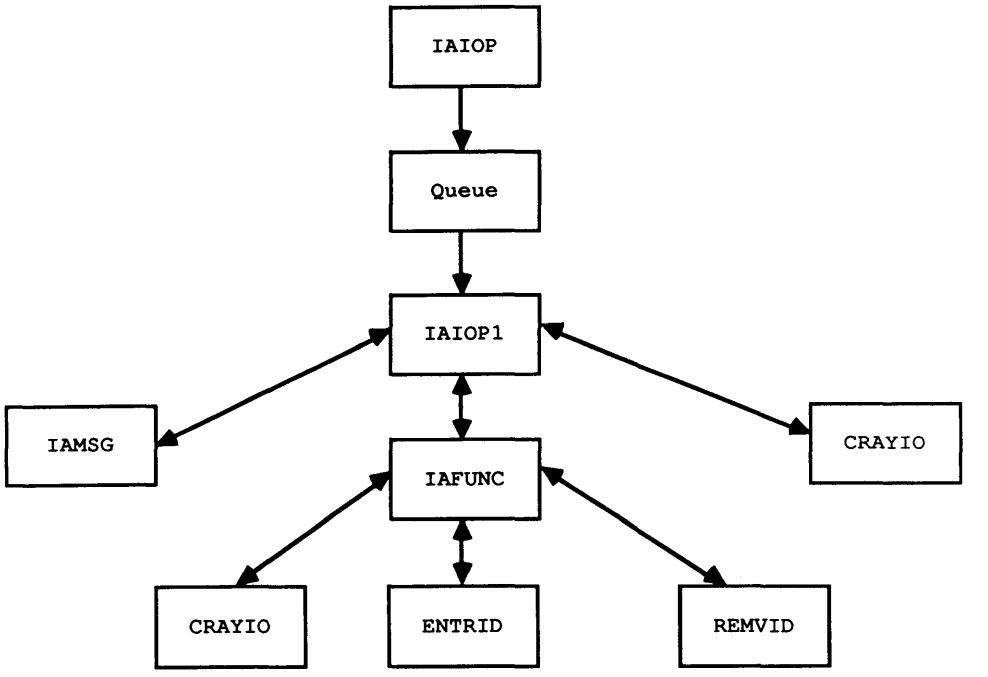


Figure 8-1. Structure of Interactive Concentrator Software

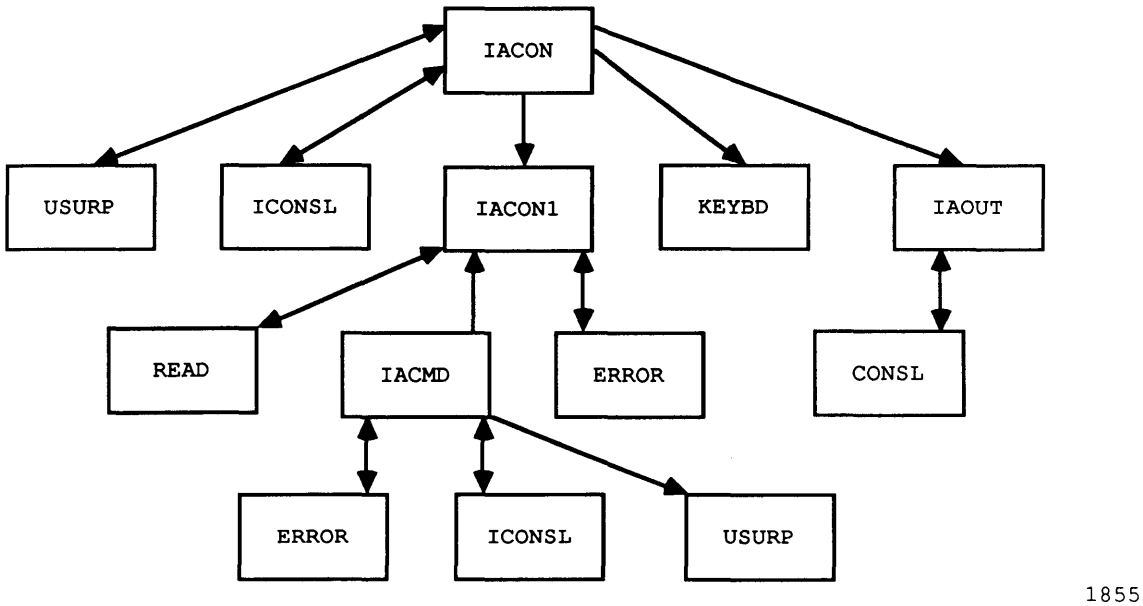


Figure 8-2. Structure of Interactive Console Software

Format:

```
IAIOP LOG [id] [tid]
```

id Optional 2-character identifier used by the mainframe to associate messages and data with this console; the default is II.

tid Optional 8-character operator station identifier; this parameter has a default of 0.

By default, each console's message buffer is checked every tenth of a second to see if it has a line of input ready to be transferred to the mainframe. The length of time between checks can be changed with the POLL command.

Format:

```
IAIOP POLL nn
```

nn Interval, in decimal tenths of a second, between checks for input

The LOGOFF command logs off the interactive concentrator. The END command logs off the interactive concentrator if it has not already been logged off and terminates it.

Formats:

```
IAIOP LOGOFF
IAIOP END
```

NOTE

The interactive concentrator must be initialized to bring up an interactive console, and all interactive consoles must be terminated to terminate the interactive concentrator.

8.1.2 IAIOP1 OVERLAY

IAIOP1 is the main control overlay of the interactive concentrator. The following stepflow describes its processes:

1. Wait for poll interval to expire or for a command function from IAIOP.

2. If it is a command function, call IAFUNC to process it.

3. If not logged on, go to step 1.

4. Begin the main body, checking each interactive console for a message to send to the mainframe.

5. Move each message bound for the mainframe from the interactive console buffers to a segment in Buffer Memory.

6. Reactivate the interactive console processes to interpret the next line of input.

7. Go to step 4 until all consoles are processed or until the segment is full.

8. Build the segment descriptor and write it to Buffer Memory.

9. Call the CRAYIO overlay to send the message to the mainframe and get a response.

10. Update the message counter.

11. Read the response LCP from Buffer Memory. If a segment is present, call IAMSG overlay to distribute the segment to the console tasks.

12. Go to step 1.

8.1.3 IAFUNC OVERLAY

Overlay IAFUNC processes three commands: LOG, LOGOFF, and END.

The stepflow for processing a LOG command is as follows:

1. Get a Buffer Memory buffer for use by the interactive concentrator while it is logged on.

2. Build the logon segment and write it to Buffer Memory.

3. Build the Descriptor Table for this ID and write it to Buffer Memory.

4. Build the logon LCP and write it to Buffer Memory.

5. Call CRAYIO overlay to send the logon message to the mainframe.

6. Prepare the LCP for interactive requests.

7. Write a message to the Kernel console confirming the logon.

The following stepflow details IAFUNC processing of a LOGOFF command.

1. Build a logoff message LCP.

2. Call CRAYIO overlay to send the message to the mainframe.

3. Release the Buffer Memory buffer used while logged on.

4. Clear the Logged On flags for all active consoles.

5. Write a message to the Kernel console confirming the logoff.

END command processing involves the following stepflow:

1. Perform all logoff processing.

2. Check all interactive consoles. If any are still active, write an error message to the Kernel console and return.

3. Release Local Memory used by the interactive concentrator and terminate the task.

8.1.4 IAMSG OVERLAY

The IAMSG overlay distributes the segment from an Interactive Reply message among the interactive consoles. The stepflow for this overlay is as follows:

1. Read the stream descriptor from Buffer Memory and get the segment address from it.

2. Begin the main loop, consisting of reading and distributing the messages.

3. Read one message from Buffer Memory.

4. If the message is a Start message, determine the corresponding console from the job name. Enter the process number in a map for finding the corresponding console on later messages and set the Logged On flag for the console. Go to step 2.

5. If the message is not a Start message, find the corresponding console from the process number.

6. Move the message for this console to its circular output buffer in Buffer Memory and update the IN pointer.

7. If the output buffer is more than three-fourths full, set the Suspend flag in this console's terminal header and set a flag to force an interactive control for the terminal.

8. Go to step 2 for all messages in the segment. Return to caller when done.

8.2 INTERACTIVE CONSOLE OVERLAYS

The overlays in the following subsections constitute the interactive console.

8.2.1 IACON OVERLAY

IACON overlay initiates an interactive console. Its stepflow is as follows:

1. Allocate Local Memory for use by the console.

2. Check to ensure the interactive concentrator is initialized. If it is not, write an error message and return.

3. Allocate a Buffer Memory buffer to hold output.

4. Enter this console into a table of consoles in the concentrator's Local Memory area.

5. Initialize local buffer addresses for this console.

6. Allocate the console to this task.

7. Create the KEYBD task to read from the keyboard.

8. Create the IAOUT task to move output from the Buffer Memory output area to the screen.

9. Go to IACON1 overlay, which is the interactive console control routine.

8.2.2 IACON1 OVERLAY

IACON1 overlay is the control task for the interactive console. It reads input from the keyboard buffer, processes it, and informs the interactive concentrator that the input is ready. The stepflow for IACON1 is as follows:

1. Call READ overlay to get a line of input.

2. If the first character is the current control character, call IACMD overlay to process the command.

3. Build the block control word (BCW) for the message.

4. Build the record control word (RCW) for the message.

5. If the console is not logged on, write an error message and go to step 1.

6. If the interactive concentrator is not logged on, write an error message and go to step 1.

7. Set flag for the interactive concentrator. A message is ready to go out.

8. Wait on queue until the interactive concentrator has sent the message in.

9. Go to step 1.

8.2.3 IACMD OVERLAY

IACMD overlay processes commands to the interactive console. All commands must be preceded by the command control character, which is a slash by default.

The following commands are available. The shortest unique string of each, underlined below, may be entered as follows.

Command	Action
ABORT	Sends an abort status to the interactive job
ATTENTION	Sends an attention status to the interactive job. (An attention status may also be sent by pressing the break key.)
BYE	Terminates the interactive console
CHANGE *c*	Changes the command control character; *c*, which can be any character, becomes the new control character.
COMMENT	Allows comments
EOF	Sends an end-of-file on the $IN dataset
LOGOFF	Logs off the interactive console
LOGON	Logs on the interactive console
STATUS	Requests Cray job status

8.2.4 IAOUT OVERLAY

Overlay IAOUT moves data from the interactive console's output buffer to the screen. It suspends output while the user is typing at the keyboard until a carriage return is entered. It also forces a resume output status to be sent to the interactive job when the output buffer becomes less than 25% full.

The stepflow for IAOUT is as follows:

1. Push onto OUTQ2 for 1 second or until output is received.
2. Check the Termination flag; if it is set, terminate.
3. Check the Hold flag; if it is set, the user is typing a line. Push onto OUTQ1 until a carriage return.
4. Check to see whether any output is ready; if not, go to step 1.

5. Read the next message from the Buffer Memory output buffer to Local Memory, unless more records exist from the previous message.

6. Reset OUT pointer for this circular buffer.

7. If Buffer Memory buffer is now less than 25% full and output is suspended, set flag to resume output.

8. Call CONSL overlay to write one record to the display.

9. Go to step 2.

9. USER CHANNEL I/O

User Channel software provides the following capabilities for connecting new devices or mainframes to the I/O Subsystem (IOS):

- Enables COS jobs executing on the mainframe to transfer data to and from user-channels connected to the IOS
- Provides easy development of software channel drivers in the IOS for networking or communications applications by site personnel
- Supports full duplex communication on IOS user-channels

User Channel software is protocol-independent and supports standard OPEN, CLOSE, READ, and WRITE operations. Special operations that are required by specific applications can be easily added to the standard set.

User Channel software resides in the Master I/O Processor (MIOP) on the IOS. It consists of the User Channel shell software supplied by Cray Research, Inc. (CRI) and various User Channel drivers.

The shell is responsible for handling requests from and responses to Central Memory, transferring data between Central Memory and channel driver, allocating all IOS resources for channel drivers, and processing hardware interrupts for user-channels.

9.1 USER CHANNEL REQUESTS

This subsection describes the processing of User Channel requests. The function codes, which begin with CR$, are received from Central Memory in F-packets on the MIOP.

9.1.1 OPEN REQUEST (CR$OPN)

CR$OPN must be the first request made for the input or output side of a user-channel. The open request names the driver overlay to be invoked for the channel.

9.1.2 READ REQUEST (CR$RD)

CR$RD transfers data from the input side of a user-channel to Central Memory. If data is already present in Buffer Memory as the result of a previous Read-Hold request, the data in Buffer Memory is transferred to Central Memory. If data is not present in Buffer Memory, the channel driver activity is called to read data from the channel. When the read completes, the data is sent to Central Memory. The response to the mainframe is then sent after the data transfer to Central Memory.

9.1.3 READ-HOLD REQUEST (CR$RDH)

CR$RDH transfers data from the input side of a user-channel to Central Memory. An additional read operation is performed on the channel with the data held in Buffer Memory. As in the Read function, any data present in Buffer Memory from a previous Read-Hold request is transferred to Central Memory first.

The channel driver is called if data is not present in Buffer Memory to satisfy the first half of the Read-Hold request. The response to the Read-Hold request is sent to the mainframe immediately after data has been transferred to Central Memory. The channel driver is then called to read data from the channel for the second half of the request. The second data is held in Buffer Memory. In this way, processing of the first data may be overlapped with the next channel read on the IOS.

9.1.4 READ-READ REQUEST (CR$RD2)

CR$RD2 transfers two data buffers to Central Memory from the input side of a user-channel. The Read-Read function is similar to the Read-Hold function except that the second data read from the channel is sent to Central Memory rather than held in Buffer Memory. In addition, the response to the mainframe is delayed until the second data buffer has been transferred. The Read-Read request does not allow the overlap of processing and I/O activity obtained by the Read-Hold request; however, the Read-Read request does reduce by 50% the interrupt overhead on the mainframe because two data reads are performed with each request.

9.1.5 WRITE REQUEST (CR$WRT)

CR$WRT transfers data to the output side of a user-channel. If data is already present in Buffer Memory as the result of a previous Write-Hold request, the data in Buffer Memory is transferred to the channel by the

driver. If data is not present in Buffer Memory, data from Central Memory is transferred to the channel by the driver. The response is sent to the mainframe after the data transfer to the user-channel is complete.

9.1.6 WRITE-HOLD REQUEST (CR$WRTH)

CR$WRTH transfers data to the output side of a user-channel. An additional buffer of data is transferred from Central Memory and held in Buffer Memory. As in the above CR$WRT function, any data present in Buffer Memory from a previous Write-Hold request is transferred to the user-channel first. If data is not present in Buffer Memory, data from Central Memory is transferred to the channel by the driver to satisfy the first half of the Write-Hold request. The second buffer of data is then transferred from Central Memory and held in Buffer Memory. The response to the Write-Hold function is then sent to the mainframe. The Write-Hold function allows buffer space in the mainframe to be freed up without waiting for the data to actually be sent out on the channel. This may be an important consideration if buffer space is limited, or is being shared for input and output on a group of channels.

9.1.7 WRITE-WRITE REQUEST (CR$WRT2)

CR$WRT2 transfers two data buffers to the output side of a user-channel. The Write-Write function is similar to the Write-Hold function with the second data from Central Memory being sent to the channel instead of held in Buffer Memory. Also, the response to the mainframe is delayed until the second data buffer has been transferred. The Write-Write function halves the interrupt overhead on the mainframe because two data writes are performed with each request.

9.1.8 DRIVER REQUEST (CR$DRV)

CR$DRV causes the channel driver to take special action that is typically protocol dependent. No data transfer is associated with the request. The response is sent to the mainframe when the channel driver has completed the request.

9.1.9 CLOSE REQUEST (CR$CLS)

CR$CLS must be the last request made for the input or output side of a user-channel. The channel driver usually terminates when a Close function is received.

9.2 SHELL ARCHITECTURE

Each user-channel pair on the MIOP is controlled by two independent shell activities, one for the input and one for the output side of the channel. Each shell activity has a corresponding driver activity created when a request is made to open a user-channel. Associated with each shell and driver activity pair is a data structure User-Channel Table (UCT) containing information needed to control the channel.

A data handling routine in the Buffer I/O Processor (BIOP) is responsible for moving data between Central Memory and Buffer Memory.

An interrupt handling routine is resident in the MIOP Kernel for processing hardware interrupts on User Channels.

Figure 9-1 shows shell architecture.

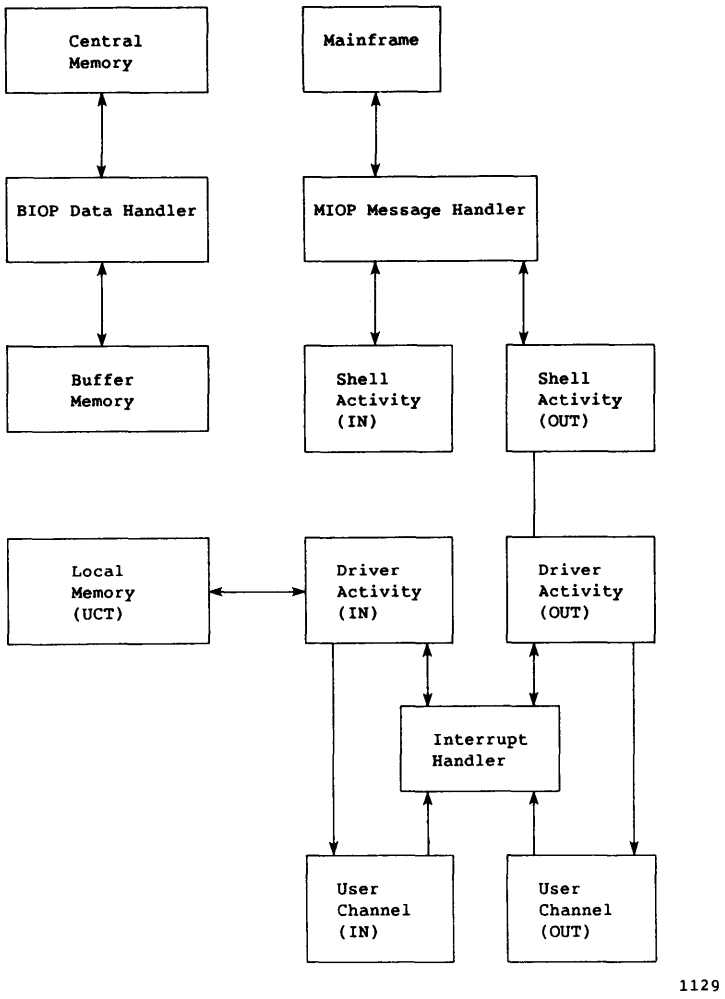


Figure 9-1. Shell Architecture

9.2.1 USER CHANNEL TABLE

The User Channel shells and channel drivers share a common data structure in MIOP Local Memory. The User Channel Table (UCT) contains a header allocated at system initialization and entries, one per input or output side of a user-channel, allocated and deallocated dynamically as channels are opened and closed. Each table entry is divided into four sections; identification and linkage information, storage used by the shell, storage used by both the shell and driver, and finally, information used by the channel driver. Use of specific fields within the UCT entry will be described in the following routines.

9.2.2 USER CHANNEL MESSAGE HANDLER

F-packet requests arriving in the MIOP Kernel are queued to the common packet handling demon (ADEM). ADEM attempts to locate the entry in the UCT referred to in the F-packet. A table entry is allocated and a shell activity created, if necessary, and the request queued to the UCSHL routine.

F-packet responses are sent to the mainframe through the SEND Kernel service request by the shell.

9.2.3 USER CHANNEL SHELL (UCSHL)

The UCSHL routine handles F-packet requests one at a time. The requested function is validated against the channel state (for example, Reads and Writes are legal only on open channels) and calls the appropriate shell subroutine for processing.

9.2.3.1 UCSHL open subroutine (UCOPN)

UCOPN saves information from the F-packet in the UCT entry for the channel being opened. The overlay specified by the driver name in the request is located and a driver activity created. The driver activity is invoked to perform any open processing required. The response to the F-packet is then returned to the mainframe.

UCOPN creates a data transfer activity in BIOP (UCXFR) and allocates a buffer descriptor table in Buffer Memory for transferring data.

9.2.3.2 UCSHL close subroutine (UCCLS)

UCCLS calls the driver activity to perform any close processing required. The response to the F-packet is then returned to the mainframe.

UCCLS terminates the data transfer activity in BIOP (UCXFR) and deallocates the Buffer Descriptor Table and any buffers containing unprocessed data in Buffer Memory.

9.2.3.3 UCSHL read subroutine (UCRD)

UCRD handles the Read, Read-Hold, and Read-Read F-packet requests. UCRD checks for data present in Buffer Memory from a prior Read-Hold function. If present, the data is transferred to Central Memory at the first address specified in the F-packet by calling UCXFR in BIOP. Excess data is truncated to the length requested in the F-packet. If data is not present in Buffer Memory, the channel driver activity is called to read data from the channel for the first length specified in the F-packet. The data is transferred to Central Memory at the first address specified in the F-packet by calling UCXFR in BIOP.

If the F-packet request was a simple Read or a Read-Hold, the length of data transferred to Central Memory is entered into the response packet and the response is sent to the mainframe.

If the F-packet request was a Read-Hold or Read-Read, the channel driver activity is called to read additional data from the channel for the second length specified in the F-packet. If the data is to be transferred to Central Memory (Read-Read), the UCXFR routine is called to send the data to the second address included in the F-packet. The length of the second data transferred to Central Memory (Read-Read) is entered into the response packet and the response is sent to the mainframe.

9.2.3.4 UCSHL write subroutine (UCWRT)

UCWRT handles the Write, Write-Hold, and Write-Write F-packet requests. UCWRT checks for data present in Buffer Memory from a prior Write-Hold function. If data is not present in Buffer Memory, it is transferred from Central Memory to Buffer Memory from the first address for the length specified in the F-packet by calling UCXFR in BIOP. The channel driver activity is then called to write the data to the channel.

If the F-packet request was a simple Write, the value for length of data transferred to the channel is entered into the response packet and the response is sent to the mainframe.

If the F-packet request was Write-Hold or Write-Write, the UCXFR routine in BIOP is called to transfer data from the second address and length in Central Memory to Buffer Memory. If the F-packet request was a Write-Hold, the response packet is then returned to the mainframe.

If the F-packet request was a Write-Write, the channel driver activity is called to write the data to the channel. The length of the second data transferred is entered into the response packet and the response is sent to the mainframe.

9.2.3.5 UCSHL driver subroutine (UCDRV)

UCDRV handles all nonstandard F-packet requests. It calls the channel driver activity to process the function and returns the response to the F-packet to the mainframe.

9.2.4 USER CHANNEL SHELL DATA HANDLER (UCXFR)

The UCXFR routine handles data transfers between Central Memory and Buffer Memory over the high-speed channel. It is called by the MIOP shell activity with a direction, Central Memory address, length, and list of Buffer Memory buffers to supply or receive data.

9.3 SHELL AND DRIVER INTERFACE

Reference was made previously in the description of the UCSHL subroutines to calling the driver activity. This subsection details the calling mechanism and parameters passed.

9.3.1 SIGNAL AND WATCH MACROS

Because the shell and driver are separate activities, the overlay CALL mechanism cannot be used for communication between them. Instead, the SIGNAL and WATCH macros are used. This implies a high degree of synchronization between the shell and driver. The shell accepts F-packet requests from the mainframe and signals the driver activity, which should be suspended watching for the next request. When the driver has completed a request it signals the shell, which has been watching for a response from the driver activity.

Several fields are allocated in the UCT entry for communication between the shell and driver activities. The SIGNAL and WATCH functions require 3 parcels of storage for queuing requests between activities. The UC@SDR and UC@SDQ fields are used for communication when the shell activity is to signal a request to a watching driver. Similarly, the UC@DSR and UC@DSQ fields are used by a driver to signal a response to the watching shell. The address of the UCT entry for the channel is passed as a parameter to the driver when the driver activity is created.

9.3.2 SHELL REQUESTS

The request codes signaled from the shell to the driver are as follows:

Code	Function
UC$OPN	Perform driver open processing
UC$CLS	Perform driver close processing
UC$RD	Read requested number of bytes from channel into supplied buffer
UC$RDL	Read last requested number of bytes from channel into supplied buffer. Driver should terminate read from channel after data has been read.
UC$WRT	Write requested number of bytes from supplied buffer to channel
UC$WRTL	Write last requested number of bytes from supplied buffer to channel. Driver should terminate write to channel after data has been written.
UC$DRV	Perform nonstandard driver processing

Additional driver-dependent parameters may be contained in the F-packet request. The original request may be referenced by the driver by using the address contained in the UC@REQ field of the UCT entry.

9.3.3 DRIVER RESPONSES

The response codes signaled from the driver to the shell are as follows:

Code	Function
UC$CMPT	The driver has successfully completed the shell request
UC$CONT	The driver has successfully completed the UC$RD or UC$WRT request. The driver is still expecting a UC$RDL or UC$WRTL request from the shell.
UC$NOOP	The driver has accepted the supplied buffer and length parameters. This is the first response to a UC$RD or UC$WRT request when the driver is double buffering.
UC$ERR	Response codes equal to or greater than this value indicate that the driver has detected an error in processing the request. The shell terminates the F-packet request processing and returns the response code to the mainframe.

9.3.4 BUFFERING

The shell performs all resource management of Local and Buffer Memory buffers. A single Local Memory buffer is always allocated for use by the channel driver on Open, Close, and Driver functions.

The shell has the ability to support either single or double buffering of data on Read and Write functions. Double buffering allows the shell to overlap movement of data to or from Buffer Memory with channel I/O by the driver because they are separate activities. The shell assumes single buffering mode unless the driver sets the UC@XBF (extra buffer) field in the UCT to 1. Typically, the buffering mode should be selected during driver open processing and should not be changed later.

Because Local Memory buffers in the IOS are 4096 bytes in length, requests to read or write large data buffers are split by the shell into several subrequests for the driver. The shell supplies a Local Memory Buffer address and length (in bytes) in the UC@SDB and UC@SDZ fields of the UCT when signaling the driver with a request. Similarly, the driver responds with a buffer address and length (in bytes) in the UC@DSB and UC@DSZ fields of the UCT when signaling the shell with a response.

9.3.5 INTERRUPT PROCESSING

The shell performs interrupt handling on user-channels in the MIOP Kernel routine IUCIO. The channel driver activity is responsible for initiating physical I/O on the channel. The driver initiates the I/O and then pushes itself onto a wait queue, UC@IWQ, in the UCT entry, by calling the TPUSH Kernel service request. A time-out should be supplied with the TPUSH call. The UC@TMO field in the UCT contains a time-out value supplied in the F-packet open request by the controlling mainframe task. The driver may use this value, if appropriate. The Interrupt Pending flag, UC@IPN, in the UCT entry should be set before the TPUSH call.

When the interrupt occurs, the IUCIO routine is entered in the Kernel. The routine locates the UCT entry corresponding to the interrupting channel. The interrupt status is read from the channel and saved in UC@IST of the UCT entry. The ending buffer address of the I/O is read and saved in UC@IBF, the Interrupt Pending flag is cleared, and the Interrupt Returned flag, UC@IRT, is set. The channel driver activity is then placed on the processor queue for execution.

The channel driver should check for a time-out by examining the result returned from the TPUSH for a value of EC$TIME. The Interrupt Returned flag, UC@IRT, should be cleared. If a time-out did not occur, the interrupt status, UC@IST, and ending buffer address, UC@IBF, should be examined for errors. Error recovery processing is the responsibility of the channel driver activity.

9.3.6 USER CHANNEL CONFIGURATION

User-channels on the IOS must be designated as type UC in the MIOP channel declaration section of the AMAP overlay. Such channels appear as type UCHN on the MIOP Kernel console CONFIG display.

9.3.7 DRIVER INSTALLATION

New drivers may be added to the IOS by replacing dummy driver overlays named UCDRV0 through UCDRV9 in the OVLNUM overlay. Driver names are limited to a maximum of 8 alphanumeric characters. The new driver overlay should then be assembled with APML and the resulting code file copied to the library of the IOS routines. New Kernel and overlay binaries should then be produced with the BIND utility.

10. NSC HYPERchannel

The I/O Subsystem (IOS) Network System Corporation (NSC) multipoint driver links a Cray mainframe and a front-end station through the NSC HYPERchannel. This driver allows multiple front-end computers to be connected through an NSC A130 adapter to one physical Master I/O Processor (MIOP) channel pair. The A130 adapter is shared by both Station Call Processor (SCP) and other protocols.

The protocol-independent interface can buffer a maximum of *n* messages with associated data on a logical path basis. This number *n* is defined at system generation time and has a default value of 4. Because there are limitations to I/O Subsystem (IOS) resources, caution must be used when the default value is exceeded. It should be noted that after the logical path buffering capability has been exhausted, any incoming messages will be discarded.

One NSC activity is associated with each physical IOP channel connected to an NSC adapter (SCP and others) and acts as an interface between the relatively complicated NSC protocol and multiple modified versions of the IOP concentrator. Because two-way alternate protocol is not enforced for each front-end station, one concentrator is set up for each logical ID connected through the NSC channel. It handles data transmission on a synchronous, point-to-point basis, which is typical of Cray protocol. The special requirement of the NSC multipoint driver is that the channel always be open for more input data, because multiple front ends can be attached to the channel.

10.1 NSC ACTIVITY INITIALIZATION

The NSC activity is initialized either by the ADEM overlay, because of a request from the mainframe, or through a command keyed at the MIOP Kernel console (refer to the I/O Subsystem (IOS) Operator's Guide for COS, publication SG-0051, or the I/O Subsystem (IOS) Operator's Guide for UNICOS, publication SG-2005).

NSC locates the Front-end Interface (FEI) Table for the ordinal specified. If it cannot find the table, it posts an error message. (The IOS operator's guides describe the NSC messages).

Next, NSC verifies that the ordinal is of the correct channel type (FEI@CT=CH$NS) and is not in one of the following states:

- Initializing (FEI@II=1)
- Terminating (FEI@TM=1)
- Active (FEI@AC=1)

If any of the preceding conditions are true, NSC posts an error message and then clears the channel pair (FEI@CH), clears the adapter, and gets the current adapter status. If the status is abnormal, an error message is posted.

NSC allocates the NIO Table for the initializing ordinal and saves the address of the NIO Table in the FEI Table (FEI@TB). The NSCIO activity is created and a completion message is posted.

10.2 NSCIO ACTIVITY

NSCIO sets the activity to active (FEI@AC=1) and locates all of the NIO buffers and queues used for accomplishing I/O. Then NSCIO allocates table space for its NIO (front-end ID) Tables and enters its idle loop.

10.2.1 NSCIO IDLE LOOP

The NSCIO idle loop is as follows:

1. NSCIO issues a wait-for-message function (NFB$WFM) to the adapter attached to its channel pair. This function sets the adapter to return an interrupt when an inbound message arrives at the A130 adapter.

2. NSCIO checks each of the following queues for requests:

   - FEI@TM≠0 indicates that NSCEND has begun termination of the NSC activity for the current ordinal. NSCIO calls NIDEND to terminate all active station IDs on the attached adapter. In addition, TERMNSC is activated to terminate protocols other than SCP. When both NIDEND and TERMNSC are complete, the NSC activity terminates.

   - The input interrupt handler sets the message cell of the wait-for-message queue (NIO@MQ) to NSC$SR in response to an interrupt following issue of a wait-for-message function (NFB$WFM). This indicates that an input message is pending. NSCIO then enters its read sequence.

- The concentrator or a logical path write activity links a write packet to the write request queue (NIO@WC) to indicate that a write to a write request is pending. NSCIO enters its write sequence.

3. NSCIO suspends itself on its wait-for-message queue and waits to be signaled by either an input interrupt or a write request from a concentrator or logical path connection. When signaled, NSCIO jumps back to step 2 of this process to check any requests.

10.2.2 WRITE SEQUENCE FOR THE PROTOCOL-INDEPENDENT INTERFACE

A typical write sequence for protocols other than SCP is as follows:

1. FNSC creates the NSCRW write activity during the assign logical path sequence.

2. After initialization, NSCRW waits for a write request from the ADEM activity.

3. When the request is received, NSCRW allocates a write request packet in Local Memory. The message is then transferred from Central Memory to MIOP Local Memory. Buffer Memory buffers are allocated, and the associated data is moved from Central Memory to Buffer Memory.

4. NSCRW puts the write request packet in the NSCIO write chain queue and signals the NSCIO overlay to execute the request. The write activity then pushes itself on a queue to wait for the write operation to be completed.

5. NSCIO completes the write operation and deallocates the buffers in Buffer Memory. NSCIO notifies NSCRW, through a pop Kernel call, that the write operation is completed.

6. NSCRW returns the ending status in the N-packet to Central Memory, releases the write request packet, and waits for the next request.

10.2.3 READ SEQUENCE FOR THE PROTOCOL-INDEPENDENT INTERFACE

A typical read sequence for protocols other than SCP is as follows:

1. FNSC creates the NSCRW read activity during the assign logical path sequence. FNSC passes parameters specifying the number of messages to buffer and their maximum data length.

2. After NSCRW initialization, a read request packet is put in the NSCIO queue. NSCRW then waits on a push queue for one of the following to occur:

    - NSCIO receives a message to satisfy the read request, and the following sequence occurs:

        - The NSCIO activity pops NSCRW.

        - NSCRW immediately allocates an additional read request packet and puts it in the NSCIO read chain queue.
        - If a CPU read request is outstanding, the driver copies the message and associated data to Central Memory, deallocates Local and Buffer Memory, and returns an ending status in the N-packet. Otherwise, NSCRW buffers the message/data (if space is available) or discards it if space is unavailable.

    - A read request is received from ADEM, and the following sequence occurs:

        - NSCRW checks to see whether any messages/data are currently being buffered for the specific logical path.

        - If messages/data are being buffered, the oldest message/data is transferred to Central Memory.

        - If no messages/data are being buffered, the driver waits for an inbound message to arrive, or an indication of a read time-out. Should a time-out occur, an error status is returned in the N-packet.

10.2.4 SCP INTERFACE LOGON SEQUENCE

The SCP interface uses a typical read/write/read operation for a logon sequence. For additional information, refer to the Front-end Protocol Internal Reference Manual, publication SM-0042.

The first read operation for the logon sequence is as follows:

1. NSCIO reads an inbound message link control package (LCP) from the NSC adapter and calls NSCID to determine the destination protocol.

2. NSCID examines the LCP content to verify that the message is for SCP. SCP requires a destination logical path of 0 and a logon message code with the appropriate segment bit count.

3. NSCID allocates Local Memory for the Front-end Station Table and its associated read/write packet. NSCID creates SCPIO to handle all communication between Central Memory and the front-end station.

4. SCPIO initializes and enters a queue that is waiting to be activated by NSCIO when the read operation is completed.

5. Before returning the read/write packet address to NSCIO and terminating, NSCID verifies that SCPIO was successfully created.

6. NSCIO puts the message in the read/write packet and reads any associated data from the adapter into Buffer Memory (the read/write packet contains pointers to all buffers containing data). NSCIO then pops SCPIO to signal the completion of the read operation.

7. SCPIO polls Central Memory with a B-packet.† SCPIO then completes the inbound data transfer.

The write operation for the logon sequence is as follows:

1. SCPIO reads the B-packet in Central Memory to obtain the address of the outbound message and associated data. SCPIO then puts the information in the read/write packet.

2. SCPIO links the read/write packet to the end of the NSCIO write chain and NSCIO begins the write operation. SCPIO enters a queue to wait for NSCIO to complete the write operation.

3. NSCIO completes the write operation and pops SCPIO to set up for a read.

For any subsequent read operations, SCPIO links a read/write packet to the NSCIO read chain and then enters a queue to wait for the next inbound message.

SCPIO has a maximum subsegment length defined by the IOS installation parameter NSCBFC, which must match the value of the COS installation parameter I@NSCBFC. SCPIO will truncate any transfers that exceed this maximum length. NSCBFC is equal to 128 kbytes by default, and it is described further in the COS Operational Procedures Reference Manual, publication SM-0043.

† The B-packet specifies the location in Central Memory in which the message and associated data are to be placed.

10.3 NSC ACTIVITY TERMINATION

NSC activity termination is accomplished by the NSCEND activity. NSCEND is created either by ADEM, in response to a CHANNEL OFF command, or through a command keyed in at the MIOP Kernel console (refer to the appropriate IOS operator's guide).

NSCEND searches for an FEI Table that corresponds to the ordinal being terminated. If no match is found, an error message is posted and NSCEND terminates.

NSCEND sets the Termination In Progress flag (FEI@TM) and waits until NSCIO responds by clearing the Active flag (FEI@AC). NSCEND then sends a clear-adapter function (NFB$CA) to the adapter. Finally, NSCEND releases the NIO Table space, posts a completion message, and terminates.

10.4 OVERLAYS

The following overlays are associated with the NSC activity. Figure 10-1 shows the NSC HYPERchannel driver overlay connections.

10.4.1 ADEM OVERLAY

The ADEM overlay routes N-packets received from Central Memory. ADEM puts a read or write request in a queue to be processed by the NSCRW job associated with the logical path. ADEM puts all other requests in a queue to be processed by the FNSC job.

10.4.2 FNSC OVERLAY

The FNSC overlay performs processing functions based on the type of the N-packet, as follows:

N-packet	FNSC Processing Functions
Open	If necessary, FNSC creates a channel table for the channel specified in the packet and links the table to the channel table chain. FNSC also allocates an open table and links it to the open table chain for the channel.

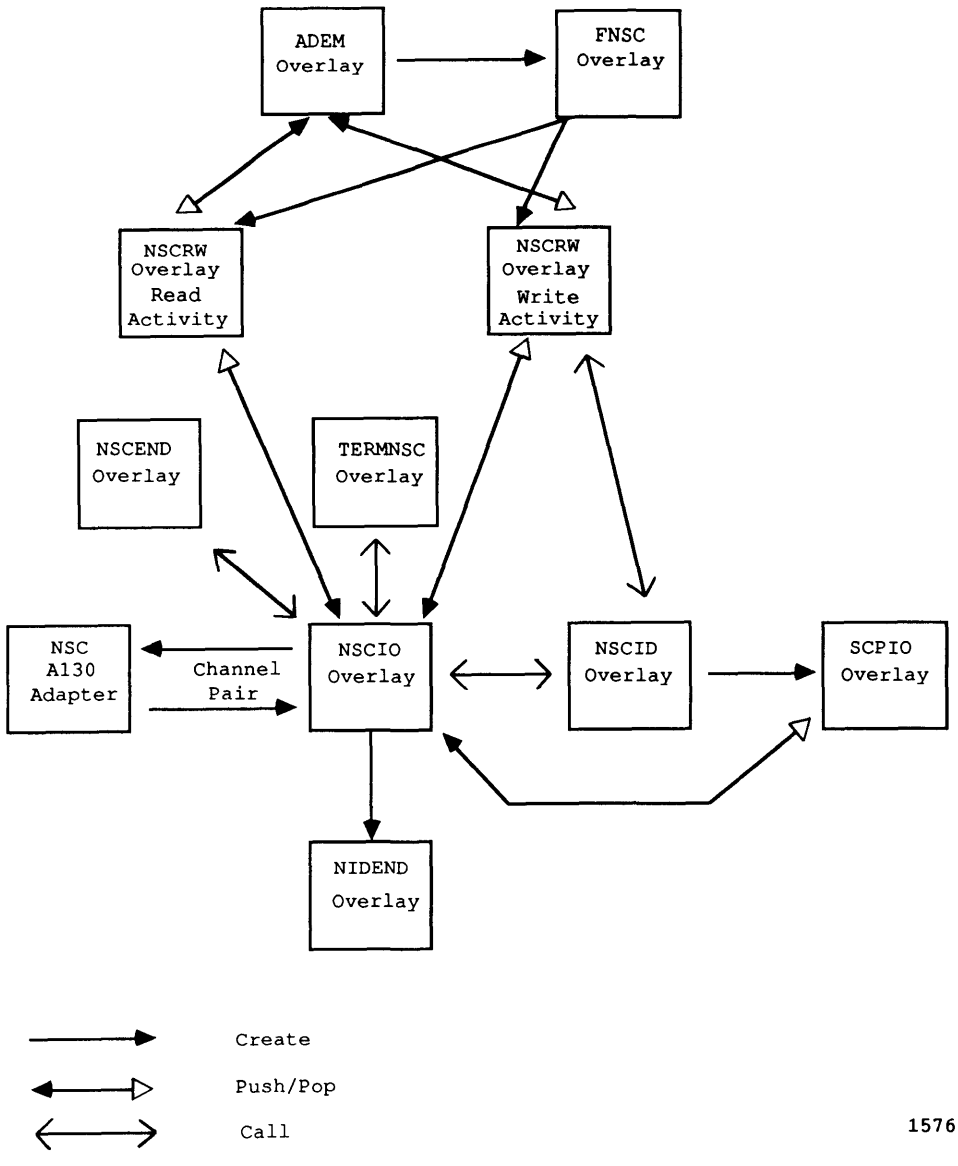


Figure 10-1. NSC HYPERchannel Driver Overlay Connections

N-packet	FNSC Processing Functions
Assign Logical Path	FNSC creates logical path tables for input and output and links them to the logical path chain for the channel number and owner identification in the packet. FNSC creates two NSCRW jobs; one to process input and one to process output.

N-packet	FNSC Processing Functions
Release Logical Path	FNSC terminates the NSCRW jobs associated with the path input and output and then releases the logical path tables.
Close	FNSC releases all logical paths associated with the channel number and owner identification specified in the packet. It also releases the open table and then the channel table (if there are no open tables linked to it).

After processing a request, FNSC returns the status in the N-packet to Central Memory. The status is one of the following:

Response Code†	Description
0	Operation completed with no errors
3	Protocol error
4	Illegal channel number
5	Illegal function
6	Illegal driver
7	Data address error
10	Data length error
11	IOS resources not available
36	Logical path not available
37	Local adapter not available
40	Message proper length error
41	Read time-out
42	Read error
43	Write time-out
44	Write error
45	No corresponding read on a loop-back write
46	Transfer length error
47	Release logical path error
50	Function code error
51	Trunk address not specified
52	Internal resource allocation error
53	Remote adapter not available
54	Driver termination in progress
55	Requested read length greater than initialized value
56	Invalid transfer length specified (write)
57	Insufficient space allocated (read)
60	Loop-back write error (read buffers exhausted)
61	Residual data present after end-of-transfer
62	Bad CPU address

† Response codes are in octal.

10.4.3 NIDEND OVERLAY

The NIDEND overlay is created by NSCIO. NIDEND terminates the NSC concentrator associated with a specific station ID, releases buffer space, and issues a LOGOFF message to the Cray mainframe.

10.4.4 NSC OVERLAY

The Kernel creates the NSC overlay during initialization or when the NSC command is entered at the MIOP Kernel console. NSC creates the NSCIO table.

10.4.5 NSCEND OVERLAY

NSCIO calls the NSCEND overlay to terminate all NSC activity. All concentrators are terminated, all buffers are released, and the I/O channel is master cleared.

10.4.6 NSCID OVERLAY

NSCIO calls NSCID to locate the read/write request packet associated with an incoming message and any associated data. For SCP protocol, the search is based on the station ID. For non-SCP protocol, the search is based on the logical path in the message proper. NSCID handles the logon LCP from the front-end station.

10.4.7 NSCIO OVERLAY

The NSCIO overlay is created by NSC. NSCIO issues I/O requests to the NSC channel pair; it also issues functions to the adapter using the A130 protocol. The NSCIO, SCPIO, and NSCRW overlays coordinate network activity.

10.4.8 NSCMSG OVERLAY

The NSCMSG routine is called by all NSC routines that require the display of a message on the MIOP Kernel console. All NSC messages consist of two lines of information. The first identifies the NSC element to which the message applies. The second contains the message.

Format:

hh:mm:ss NSC: CONCENTRATOR *x* [ORDINAL *y*]
message

hh:mm:ss	Time in hours, minutes, and seconds
x	NSC concentrator ordinal number in octal
y	Front-end ordinal number in octal
message	See the appropriate IOS operator's guide for a list of NSC messages

10.4.9 NSCRW OVERLAY

FNSC creates the NSCRW overlay to handle any non-SCP protocol. The NSCRW read activity buffers incoming messages and data. Hooks are provided to run adapter diagnostics.

10.4.10 SCPIO OVERLAY

NSCID creates the SCPIO overlay to process network messages and associated data through the SCP protocol handler for the front-end stations. NSCID creates an SCPIO activity for each unique station ID.

10.4.11 TERMNSC OVERLAY

The TERMNSC overlay terminates all logical path connections.

10.5 ERROR RECOVERY

Separate error recovery schemes exist for SCP protocol and for the protocol-independent portion of the driver.

10.5.1 ERROR RECOVERY FOR SCP PROTOCOL

The error recovery scheme for SCP protocol consists of the following:

- Driver input/read operations
- Driver output/write operations

The HYPERchannel driver does not generate front-end driver (FED) error codes, so the front-end stations do not receive FED error codes 330, 331, 332, and 333.† In addition, the driver does not return an FED error code of 307 to SCP when a write error occurs.

10.5.1.1 Driver input/read operations

The driver input/read operations are as follows:

- Error recovery is not invoked.
- The driver does not send an error code to SCP or to the front-end station; instead, it waits for the next inbound message.
- The front-end station detects the error as a software time-out. For information on error recovery, refer to the Front-end Protocol Internal Reference Manual, publication SM-0042.

10.5.1.2 Driver output/write operations

The driver output/write operations are as follows:

- The device tries to execute each write request for a period of 30 seconds before aborting. Each attempt to complete a write request consists of two retries, separated by a short time delay. If both retries are unsuccessful, the write request is requeued at the end of the NSCIO write chain. Further attempts will be made after other requests have had an opportunity to complete.
- If the write operation is not completed, SCP is not notified, and the front-end station detects this as a software time-out.†

† The FED error codes are in octal. For additional information, refer to the Front-end Protocol Internal Reference Manual, publication SM-0042.

10.5.2 ERROR RECOVERY FOR THE PROTOCOL-INDEPENDENT INTERFACE

The error recovery scheme for the protocol-independent interface consists of the following:

- Driver input/read operations
- Driver output/write operations

10.5.2.1 Driver input/read operations

The driver input/read operations are as follows:

- Error recovery is not invoked.
- Error status is returned.

10.5.2.2 Driver output/write operations

The driver output/write operations are as follows:

- Each attempt to complete a write request consists of two retries, separated by a short time delay. If both retries are unsuccessful, the write request is requeued at the end of the write chain. Further attempts are made after other write requests have had a chance to complete. A write request returns an error if the operation has not completed and the write timer has expired. (Timer value is set by the Assign Logical Path command).

NOTE

The driver returns an error immediately on any unrecoverable conditions, such as an attempt to send a message to a nonexistent adapter.

10.6 CHANNEL/ID ORDINAL DESCRIPTION

Any IOP station, concentrator, or front-end station activity known to COS has a corresponding Channel Extension Table (CXT) entry in the COS Executive. Front-end station activities are those that handle station IDs logged on to COS over an FEI, NSC, or VMEbus connection (for more information on the VMEbus driver, refer to section 13, VMEbus (FEI) Driver).

See the COS Table Descriptions Internal Reference Manual, publication SM-0045, for information about the CXT. A CXT entry is directly related to the channel ordinal number assigned to its activity on the IOP. For example, if the IOP station is assigned to ordinal 1, the first CXT entry is used.

Any master operator station logged on to COS can initiate the concentrator or front-end station on the IOP by turning on its associated CXT entry. For example, if a front-end station activity is assigned to ordinal 5, the master operator station command CHANNEL 4,5 ON initiates that IOP front-end station activity. The IOP is on the mainframe's physical channel 4, and 5 is the requested ordinal.

Likewise, any master operator station can terminate a particular ID that is logged on through an IOP front-end station activity, using the same command: CHANNEL 1,*ordinal* OFF (*ordinal* is the ordinal number assigned to the activity). Ordinals assigned to a front-end activity such as a concentrator, VMEbus driver, or NSC driver can be determined by an examination of the MIOP CONFIG display. Ordinals assigned to a logged-on ID can be determined by an examination of the master operator station's LINK display.

Only channel ordinals assigned directly to a front-end activity (currently only the concentrator, VMEbus activity, and an NSC activity) can be initiated or terminated by the CHANNEL command. The interactive station and operator station on the IOP require the respective Kernel or station commands.

If a channel ordinal is turned off, communication between the mainframe and the IOP associated with that CXT ordinal is disabled. It can be reenabled with the CHANNEL ON command.

Two Kernel commands control the initiation or termination of NSC-related activities: NSC *ordinal* and NSCEND *ordinal*. NSC *ordinal* initiates the NSC activity on the IOP channel associated with the specified ordinal; NSCEND *ordinal* terminates it.

Two Kernel commands control initiation or termination of VMEbus-related activities: VME *channel mode* and VMEND *channel* (*channel* is the physical input channel number of the low-speed channel pair connecting the IOS to the VMEbus that is to be initiated. *mode* is the mode of execution; graphics and networking are the valid modes). VME *channel mode* initiates the VMEbus driver activity on *channel*. The activity is brought up in *mode*. VMEND *channel* terminates the VMEbus (FEI) Driver activity on *channel*.

The following are the default channel ordinal assignments in the IOP; they can be changed to fit the needs of individual systems.

Ordinal	Descriptor
1	IOP station 0
2	IOP station 1
3	Concentrator 0(CONC)
4	Concentrator 1(CONC)
5	Interactive concentrator
6	VMEbus activity
7	NSC activity

Ordinal assignments to IDs begin with ordinal number 8 by default. The number of CXT entries equals 7 plus the number of logical IDs coming over the NSC and VMEbus channels combined. The number of ordinals available for assignment is directly related to the number of CXT entries configured on the Cray operating system; that is, the number of CXTs equals the number of channel ordinals configured.

11. FRONT-END INTERFACE LOGICAL PATH ACTIVITY

The Front-end Interface (FEI) logical path driver provides an FEI connection for UNICOS. This connection parallels the NSC logical path connection by making use of the F/N-packet as defined in the COS Table Internal Reference Manual, publication SM-0045. It allows front-end stations to communicate with the UNICOS Station Call Processor (USCP) under UNICOS by using the SCP protocol.

Two FEI logical path activities, FEIR and FEIW, are associated with each physical IOP channel pair connected to an FEI device. Only one logical path per channel pair is supported.

11.1 FEI LOGICAL PATH ACTIVITY INITIALIZATION

The FEI logical path activity is initialized (or created) when an Assign Logical Path command is processed by FNSC. At this point, FNSC verifies that the channel pair is connected to the FEI device, and that this device is not being used by COS in the Guest Operating System (GOS) environment. Equally important, the activity ensures that the channel type is correct. If any of these conditions are not correct, an error message is posted.

The FEI logical path activity allocates the F/N table and saves the address in the FEI table (FEI@TB). The FEIR and FEIW overlays are created, and a completion message is posted.

Currently, this connection is used solely by USCP. Refer to the Front-end Protocol Internal Reference Manual, publication SM-0042, for more protocol information.

11.2 FEI LOGICAL PATH ACTIVITY TERMINATION

The FEI logical path activity is terminated when FNSC receives a Release Logical Path command from the CPU.

11.3 OVERLAYS

The following overlays are associated with the FEI logical path activity. Refer to figure 11-1 for the FEI logical path driver overlay connections.

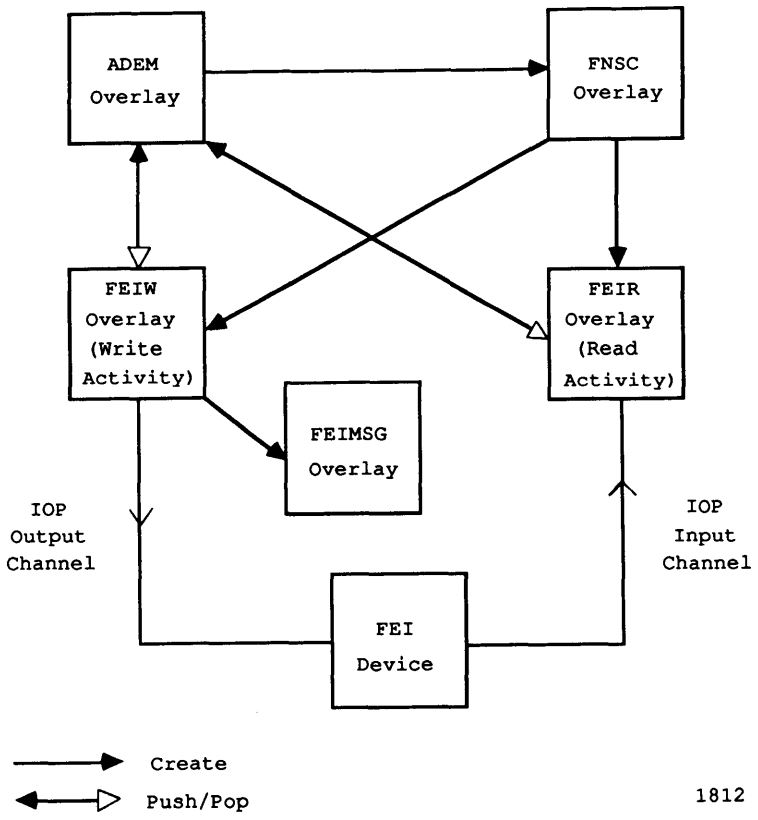


Figure 11-1. FEI Logical Path Overlay Connections

11.3.1 ADEM OVERLAY

The ADEM overlay routes N-packets received from the CPU. ADEM puts a read or write request in a queue to be processed by the FEIR and FEIW activities associated with the logical path. ADEM puts all other requests in a queue to be processed by the FNSC activity.

11.3.2 FNSC OVERLAY

The FNSC overlay performs processing functions based on the type of the N-packet, as follows:

N-packet	FNSC Processing Functions
Open	If necessary, FNSC creates a channel table for the channel specified in the packet and links the table to the channel table chain. FNSC also allocates an open table and links it to the open table chain for the channel.
Assign Logical Path	FNSC creates logical path tables for input and output and links them to the logical path chain for the channel number and owner identification in the packet. FNSC creates FEIR to process input and FEIW to process output.
Release Logical Path	FNSC terminates the FEIR and FEIW jobs associated with the path input and output and then releases the logical path tables.
Close	FNSC releases all logical paths associated with the channel number and owner identification specified in the packet. It also releases the open table and then the channel table (if there are no open tables linked to it).

After processing a request, FNSC returns a status in the N-packet to Central Memory.

11.3.3 FEIR OVERLAY

The FEIR overlay processes the read requests from ADEM and controls the input channel of the FEI device. FEIR sends data to Central Memory location and responds to all N-packet read requests.

11.3.4 FEIW OVERLAY

The FEIW overlay processes the write requests from ADEM and control the output channel of the FEI device. FEIW reads data from Central Memory and responds to all N-packet write requests.

11.3.5 FEIMSG OVERLAY

The FEIMSG overlay generates the FEI logical path information messages for the MIOP console. It is created by FEIW.

12. HSX CHANNEL INTERFACE

The I/O Subsystem (IOS) HSX channel driver software provides the following capabilities to support the CRI HSX High-speed External Communications Channel:

- Provides a protocol-independent driver for HSX channels connected to the IOS
- Supports full-duplex communication on HSX channels (enabling loop-back testing)
- Allows data transfer between HSX channels and any of three target memories: Central Memory, SSD Memory (through the IOS backdoor), and the Buffer Memory Resident (BMR) portion of Buffer Memory

The driver provides the ability to read or write discontiguous target memory buffers in the same HSX data block by giving the mainframe control of the End-of-block channel signal. The driver software supports the configuration of an HSX channel on a Buffer I/O Processor (BIOP), Disk I/O Processor (DIOP), or an Auxiliary I/O Processor (XIOP). The IOS HSX driver requires a standard Cray 100-Mbyte channel connecting the IOS to the desired target memory on the same IOP as the HSX channel. If SSD Memory is chosen as the target memory, the SSD I/O buffers must begin on a 64-word boundary, and they must be an integral multiple of 64 words in length.

The HSX driver software can coexist with other channel drivers in the same IOP, including the on-line tape and disk drivers; however, performance drops on all channels when the HSX channel and other channel types are active simultaneously.

12.1 HSX CHANNEL REQUESTS

This subsection describes the processing of HSX channel requests. The function codes, which begin with HSF$, are received from the mainframe in H-packets (destination ID of RQ$HSX). See subsection 2.14, MIOP-mainframe Communication Channel, for more information about packet disposition. See the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for a description of the HSX request packet.

The IOS HSX driver supports OPEN, READ, WRITE, CONTROL, and CLOSE requests from the mainframe. At the completion of each request, the driver returns status information to the mainframe in the original H-packet.

12.1.1 OPEN REQUEST (HSF$OPEN)

HSF$OPEN must be the first request made for the input or output side of the HSX channel. Each side of the channel must be opened by a separate request. The OPEN request causes the HSX driver to load the input or output interrupt handler into Local Memory, and to allocate Local Memory I/O buffers for the duration of the time the channel is open.

12.1.2 READ REQUEST (HSF$READ)

HSF$READ transfers data from the input side of the HSX channel to the target memory address specified in the request. One or two target memory buffers may be supplied in the request. The response to the mainframe is sent after all the data is transferred to the target memory buffer(s).

12.1.3 WRITE REQUEST (HSF$WRIT)

HSF$WRIT transfers data from the target memory address specified in the request to the output side of the HSX channel. One or two target memory buffers may be supplied in the request. The response to the mainframe is sent after all the data is transferred from the target memory buffer(s) to the channel.

12.1.4 CONTROL REQUEST (HSF$CNTL)

HSF$CNTL causes the HSX driver to take special action regarding HSX channel control. No data transfer is associated with the request. Subfunction codes that begin with HSS$ specify the necessary action.

12.1.4.1 Set parameters (HSS$SET)

HSS$SET causes the HSX driver to set software execution control parameters in the HSX control table. The parameters that may be specified are a software channel time-out value in tenths of a second, and a debug mode flag. If no HSS$SET request is issued, or if a 0 time-out value is set in an HSS$SET request, the IOS driver default value is used. Setting the debug mode flag causes the HSX driver to allocate a

Buffer Memory buffer used to simulate HSX channel I/O on subsequent requests. Clearing the debug mode flag causes the HSX driver to release the Buffer Memory buffer. See subsection 12.3, Debug Mode, for more information about the debug mode.

12.1.4.2 Send interrupt (HSS$SNDI)

HSS$SNDI causes the HSX driver to send either a clear pulse signal on the output side of the channel or an exception signal on the input side of the channel. These signals may be used as part of a user-defined protocol.

12.1.4.3 Receive interrupt (HSS$RECI)

HSS$RECI causes the HSX driver to wait for either a clear pulse signal on the input side of the channel or an exception signal on the output side of the channel. These signals may be used as part of a user-defined protocol.

12.1.5 CLOSE REQUEST (HSF$CLOS)

HSF$CLOS must be the last request made for the input or output side of the HSX channel. A close request causes the HSX driver to unload the input or output interrupt handler, and to release the Local Memory I/O buffers allocated when the channel was opened.

12.2 HSX DRIVER ARCHITECTURE

The HSX driver software consists of one demon overlay (HCOM) and two interrupt handler routines; one for the input side (HSXI) and another for the output side (HSXO) of the channel. A common data structure called the HSX Control Block (HCB) is shared by all of these routines. Refer to the IOS Table Descriptions Internal Reference Manual, publication SM-0007, for a description of the HCB.

12.2.1 HSX DEMON OVERLAY (HCOM)

HCOM is responsible for all activities related to the HSX channel, except for the handling of hardware interrupts. These activities include receiving and processing HSX requests from the mainframe, starting channel I/O, and restarting channel I/O if the interrupt handler temporarily halts I/O for any reason.

12.2.2 HSX INPUT INTERRUPT HANDLER (HSXI)

HSXI is the heart of the HSX read activity. After HCOM starts the channel input, HSXI performs all necessary actions to keep I/O functioning until the request is completed.

When an interrupt is detected on the HSX input channel, HSXI checks for errors, restarts channel input to the next Local Memory buffer, and starts the high-speed channel write to transfer the data just received to the target memory. HSXI sends the response to the mainframe after waiting for the completion of the last data transfer to the target memory.

If HSXI detects that other software activities are waiting, it temporarily suspends HSX I/O to allow those routines to perform their tasks and it activates HCOM to restart the channel.

12.2.3 HSX OUTPUT INTERRUPT HANDLER (HSXO)

HSXO is the heart of the HSX write activity. After HCOM starts the channel output, HSXO performs all necessary actions to keep I/O functioning until the request is completed.

When an interrupt is detected on the HSX output channel, HSXO checks for errors, restarts channel output from the next Local Memory buffer, and starts the high-speed channel read to transfer the next data from the target memory to Local Memory. HSXO sends the response to the mainframe when the last data has been written to the channel.

If HSXO detects that other software activities are waiting, it temporarily suspends HSX I/O to allow those routines to perform their tasks and it activates HCOM to restart the channel.

12.2.4 BUFFERING

HCOM performs all resource management of Local Memory buffers for the HSX driver. At channel open time, HCOM allocates two Local Memory buffers of equal length to support the input or output channel activity. Double buffering allows the driver to overlap channel I/O with data movement between the target memory and Local Memory.

Because requests for the HSX channel are typically for data blocks larger than the Local Memory buffer size, the driver must split the I/O into several smaller requests to the channel; therefore, the size of the Local Memory buffers directly affects HSX channel performance. The larger the buffers, the less interrupt overhead and the better the performance; therefore, HCOM attempts to allocate buffers that are larger than normal.

Local Memory buffers in the IOS are normally 4096 bytes in length. HCOM tries to find an integral number of these buffers that are contiguous. The number of 4096-byte buffers to allocate is controlled by $APTEXT, which defines a minimum and maximum buffer size. The minimum size is specified by HSX$IBMN for the input channel and HSX$OBMN for the output channel. The maximum size is specified by HSX$IBMX and HSX$OBMX for the input and output channels, respectively.

If the maximum number of contiguous buffers cannot be found, HCOM accepts a smaller number. If the minimum number cannot be found, an error response is sent to the mainframe indicating insufficient resources to open the channel.

12.3 DEBUG MODE

A debug mode is provided to facilitate software testing when no HSX channel hardware is available. In debug mode, the IOS reads or writes data to a Buffer Memory buffer instead of the HSX channel. Debug mode does not support loop-back testing; that is, data is not preserved in Buffer Memory from a write request to a read request. Debug mode does, however, allow either read or write performance testing.

Debug mode may be enabled for only one side of the channel at a time. This is because the Buffer Memory channel is a half-duplex channel and there is no way to distinguish an input interrupt from an output interrupt. Debug mode is controlled by a flag in the HSR$CNTL request packet with a subfunction code of HSS$SET.

12.4 OVERLAY LISTING

The HSX channel overlays are grouped together with the APML list identifier of $HSX.

12.5 ERROR PROCEDURES

The error procedures performed by the IOS HSX driver are intended to be very simple so that a variety of protocols can use the same hardware driver. Each separate protocol may implement additional error procedures (such as retries) as needed at the protocol level of the software.

12.5.1 INPUT ERRORS

The following error procedures are performed by the IOS when unusual conditions are detected during HSX input channel activation.

12.5.1.1 Clear pulse received (HST$CLR)

When a clear pulse is received, the IOS performs the following command sequence:

1. Reset input channel control logic.
2. Wait 250 ns as required by hardware-defined protocol.
3. Send exception pulse to transmitting device.
4. Return status and transfer length to mainframe.

12.5.1.2 Multiple bit error (HST$DATA)

When a multiple bit error is received, the IOS performs the following command sequence:

1. Continue reading until end-of-block.
2. Send exception pulse to transmitting device (before clearing enable block hardware signal).
3. Wait 250 ns as required by hardware-defined protocol.
4. Reset input channel control logic (clearing enable block signal).
5. Return status and transfer length to mainframe.

12.5.1.3 Data overrun error (HST$OVER)

When a data overrun error is received, the IOS performs the following command sequence:

1. Continue reading until end-of-block.
2. Send exception pulse to transmitting device (before clearing enable block hardware signal).
3. Wait 250 ns as required by hardware-defined protocol.
4. Reset input channel control logic (clearing enable block signal).
5. Return status and transfer length to mainframe.

12.5.1.4 Long block error (HST$LONG)

When a long block error is detected, the IOS performs the following command sequence:

1. Return status and transfer length to the mainframe.

2. If indicated in the mainframe request, continue to read until end-of-block and throw away the data to drain the channel.

12.5.1.5 Software time-out (HST$TMO)

When a software time-out is detected, the IOS performs the following command sequence:

1. Reset input channel control logic.

2. Return status and transfer length to mainframe.

12.5.1.6 Device not present (HST$NDEV)

When an error is received because a device is not present, the IOS performs the following command sequence:

1. Reset input channel control logic.

2. Return status and transfer length to mainframe.

12.5.1.7 Short block error (HST$SHRT)

When an unexpected end-of-block signal is received (indicating a short block), the IOS returns the status and transfer length to the mainframe.

12.5.2 OUTPUT ERRORS

The following error procedures are performed by the IOS when unusual conditions are detected during HSX output channel activation.

12.5.2.1 Exception pulse received during transfer (HST$XDT)

When an exception pulse is received during a transfer, the IOS performs the following command sequence:

1. Continue writing until end-of-block.
2. Reset output channel control logic.
3. Return status and transfer length to mainframe.

12.5.2.2 Exception pulse received while channel idle (HST$XFT)

When an exception pulse is received while the channel is idle, the IOS performs the following command sequence:

1. Reset output channel control logic.
2. Return status and transfer length to mainframe.

12.5.2.3 Receiving device aborted (HST$ABRT)

When a device has aborted, the IOS performs the following command sequence:

1. Reset output channel control logic.
2. Return status and transfer length to mainframe.

12.5.2.4 Software time-out (HST$TMO)

When a software time-out occurs, the IOS performs the following command sequence:

1. Reset output channel control logic.
2. Return status and transfer length to mainframe.

12.5.2.5 Device not present (HST$NDEV)

When an error is received because a device is not present, the IOS performs the following command sequence:

1. Reset output channel control logic.
2. Return status and transfer length to mainframe.

12.6 SPECIAL SEQUENCES

Special command sequences performed by the IOS HSX driver are described in this subsection. Each separate protocol may use the special sequences as appropriate for its communication needs.

12.6.1 INPUT SEQUENCES

The following command sequences are defined to allow special control of the HSX input channel.

12.6.1.1 Send exception pulse (HSS$SNDI)

When an exception pulse is sent, the IOS performs the following command sequence:

1. Reset input channel control logic.
2. Send exception pulse to transmitting device.
3. Return status to mainframe.

12.6.1.2 Wait for clear pulse (HSS$RECI)

Waiting for a clear pulse causes the IOS to perform the following command sequence:

1. Wait for clear pulse from transmitting device.
2. Reset input channel control logic.
3. Wait 250 ns as required by hardware-defined protocol.
4. Send exception pulse to transmitting device.
5. Return status to mainframe.

12.6.2 OUTPUT SEQUENCES

The following command sequences are defined to allow special control of the HSX output channel.

12.6.2.1 Send clear pulse (HSS$SNDI)

When a clear pulse is sent, the IOS performs the following command sequence:

1. Reset output channel control logic.
2. Send clear pulse to receiving device.
3. Wait for exception pulse to be returned.
4. Reset output channel control logic.
5. Return status to mainframe.

12.6.2.2 Wait for exception pulse (HSS$RECI)

Waiting for an exception pulse causes the IOS to perform the following command sequence:

1. Wait for exception pulse from receiving device.
2. Reset output channel control logic.
3. Return status to mainframe.

13. VMEBUS (FEI-3) DRIVER

The I/O Subsystem (IOS) VMEbus (also called the FEI-3) driver allows a VMEbus-based front-end processor connected to a CRI VMEbus interface to communicate with a Cray computer system through the IOS. The IOS is physically connected to the remote VMEbus interface board by a CRI low-speed channel operating in either 6-Mbyte or 12-Mbyte mode.

The VMEbus driver allows the simultaneous use of multiple application protocols, and it uses the same mechanism for routing traffic between the Cray computer system and multiple remote addresses, as does the IOS NSC driver. Routing is done on a logical path basis, where multiple logical paths may be assigned for each physical low-speed channel.

The interface used between the IOS and UNICOS is the N-packet; the interface to COS is the B-packet for the SCP protocol and the F-packet for other protocols.

Because of the nature of the low-speed channel and the software running in both the IOS and the Cray computer system, applications using the VMEbus interface must transfer data lengths that are multiples of a Cray word (64 bits).

13.1 N-PACKET INTERFACE

The N-packet interface provides the following capabilities for support of the VMEbus driver interface:

- Provides full-duplex access to an IOS low-speed channel
- Shares one IOS low-speed channel for use among multiple Cray jobs running different applications and protocols
- Allows one Cray job to access multiple IOS low-speed channels
- Allows one Cray job to assign multiple logical paths on one IOS low-speed channel
- Queues multiple read and write requests generated by a Cray job
- Buffers incoming messages in the IOS until read by a Cray job

The N-packet interface restricts a VMEbus driver interface by allowing a maximum transfer length of 16,384 Cray words (131,072 bytes). This value can be changed by using the IOS configuration parameter NSCBFC and the COS configuration parameter I@NSCBFC (on COS systems only) The same value is used for both NSC and VMEbus interfaces operating concurrently in a single IOS.

For detailed information on the N-packet, see the IOS Table Descriptions Internal Reference Manual, publication SM-0007.

13.2 DRIVER OVERLAYS

Figure 13-1 shows the relationship of overlays and the IOS Kernel in the VMEbus driver. In the figure, all overlays above and including the NSCRW and SCPIO overlays, constitute the N-packet and B-packet interface portion of the driver. These overlays are used in the VMEbus and the NSC HYPERchannel interface software. Those below the NSCRW and SCPIO overlays constitute the portion of the VMEbus driver software that is specific to the functioning of a low-speed channel.

The following overlays are associated with the IOS VMEbus driver. See figure 13-1 for overlay connections.

13.2.1 ADEM OVERLAY

The ADEM overlay routes all N-packets received from Central Memory. F-packets received from COS that have the proper value (CR$NSC) in the IOP field of the packet will be handled as N-packets from this point on in the driver. ADEM puts a read or write request N-packet in a queue to be processed by the NSCRW activity associated with the logical path. ADEM places all other N-packet request types (Open, Assign Logical Path, Release Logical Path, and Close) in a queue to be processed by the FNSC activity.

13.2.2 FNSC OVERLAY

The FNSC overlay performs functions based on the type of the N-packet, as described below. After the function is attempted, the status is returned to Central Memory.

Type	Function
Open	If necessary, FNSC creates a channel table for the channel specified in the N-packet and links this table to the channel table chain. FNSC also allocates an open table and links it to the open table chain for the specified channel.

Type	Function
Assign Logical Path	FNSC creates two logical path tables (one for input and one for output) and links them to the logical path chain for the channel number and owner identification given in the N-packet. FNSC creates two NSCRW activities; one to process input and communicate with the VMERD activity, and one to process output and communicate with the VMEWT activity.
Release Logical Path	FNSC terminates the NSCRW activities associated with the specified path and then releases the related logical path tables.
Close	FNSC releases all logical paths associated with the channel number and owner identification specified in the N-packet, and FNSC releases the open table. Then, if there are no more open tables linked to this channel table, the channel table is released.

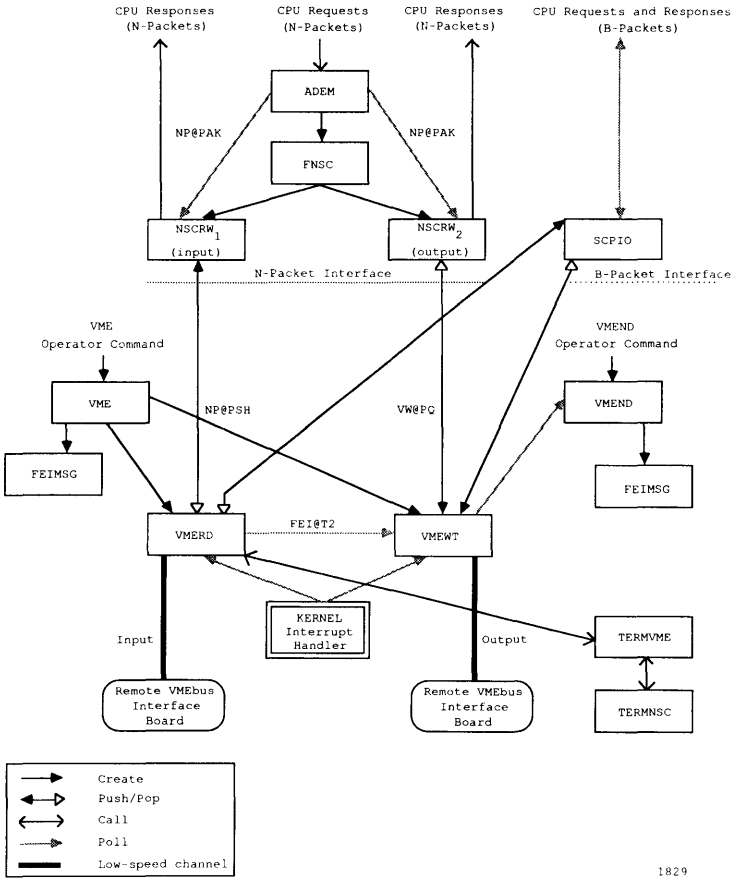


Figure 13-1. VMEbus Driver Overlay Connections

13.2.3 NSCRW OVERLAY

FNSC creates the NSCRW overlay to handle the routing of messages and data for a given logical path. There are actually two activities created, each one an identical copy of the NSCRW overlay. One copy handles strictly input traffic and buffers incoming messages and data. The other copy handles strictly output traffic.

13.2.4 VME OVERLAY

The VME overlay is created when the VME command is entered at the Master I/O Processor (MIOP) Kernel console. This command must be entered before any VMEbus interface traffic is begun. VME performs channel pair initialization and creates the two activities that physically drive the channels, VMERD and VMEWT. A message relaying the status of the initialization attempt is sent to the MIOP Kernel console.

13.2.5 VMEND OVERLAY

The VMEND overlay is created when the VMEND command is entered at the MIOP Kernel console. It may also be created by VMERD or VMEWT if a catastrophic error occurs. VMEND terminates all driver software for the VMEbus interface. A message relaying the status of the termination attempt is sent to the MIOP Kernel console.

13.2.6 FEIMSG OVERLAY

The FEIMSG overlay is created by either VME or VMEND to display an MIOP Kernel console message, and it is used by the FEI-logical path interface software. FEIMSG terminates itself after the message is displayed.

13.2.7 VMERD OVERLAY

VMERD physically controls the low-speed input channel. It passes the received messages and data to the NSCRW activity responsible for the logical path on which the input was received. A status of the attempted read is also returned to the NSCRW activity.

13.2.8 VMEWT OVERLAY

This overlay physically controls the low-speed output channel. It receives output from the various NSCRW activities and returns a status to a NSCRW activity after it has attempted the requested write.

13.2.9 TERMVME OVERLAY

This overlay is called by VMERD to terminate all logical paths on the given channel pair.

13.2.10 TERMNSC OVERLAY

This overlay is called by TERMVME to terminate the NSCRW activities responsible for the logical paths on the given channel pair.

13.2.11 NSCID OVERLAY

NSCIO calls the NSCID overlay to locate the Read/Write Packet associated with an incoming message and any associated data. For the SCP protocol, the search is based on the station ID in the LCP. NSCID processes the Logon LCP from the front-end station and creates an SCPIO activity to process future LCPs for that station. For other protocols, the search is based on the destination logical path in the message proper.

13.2.12 SCPIO OVERLAY

The SCPIO overlay processes the messages (LCPs) and data involved with one station ID. NSCID creates an SCPIO overlay for each unique station ID that logs on to COS through the SCP protocol.

13.3 READ AND WRITE REQUESTS FLOW DESCRIPTIONS

The VMEbus driver is told how many sectors (512_{10} Cray words) of associated data follow an input Message Proper by a count field in the Message Proper. Likewise, the VMEbus driver will set this field in all output Message Propers for the remote driver's use. This field is currently necessary to run the interface over 6-Mbyte low-speed channels. This subsection provides the stepflow for read and write requests.

13.3.1 READ REQUEST SEQUENCE

A typical read sequence is as follows. All references to NSCRW pertain only to the "read" copy of this overlay.

1. NSCRW has initially issued a read request RWP packet to VMERD; this is the idle state of the driver.

2. NSCRW waits on a push queue for one of the following to occur:

   - VMERD receives a message to satisfy the read request and the following occurs:

     - VMERD pops NSCRW into activation.

     - NSCRW immediately initiates another read RWP request packet and puts it on VMERD's read chain queue.

     - If a read request N-packet from the Cray mainframe is outstanding, NSCRW copies the message proper and associated data to the Cray mainframe, deallocates Local and Buffer Memory resources, and returns the ending status in the N-packet to the mainframe. Otherwise, NSCRW buffers the message and data (if space is available), or discards the message and data (if space is not available).

   - A read request N-packet is received from ADEM and the following occurs:

     - NSCRW checks to see whether any messages are currently being buffered for the specified path.

     - If messages (and associated data, if applicable) are currently buffered, the oldest message (and its data) is transferred to Central Memory.

     - If no messages are buffered, NSCRW waits for a specified period of time for an indication from VMERD that an appropriate message has arrived. If an appropriate message does arrive, it (and its data) is transferred to the mainframe with the read operation's ending status also returned in the N-packet. Otherwise, a time-out error status is returned to the mainframe in the N-packet.

3. NSCRW returns to step 2.

13.3.2 WRITE REQUEST SEQUENCE

A typical write sequence is as follows. All references to NSCRW pertain only to the "write" copy of this overlay.

1. NSCRW waits for a write request from the ADEM activity.

2. When the request is received, NSCRW allocates a write request packet in MIOP Local Memory. The message proper is then transferred from Central Memory to Local Memory. Buffer Memory buffers are allocated, and the associated data is transferred from Central Memory to these buffers.

3. NSCRW puts the write request packet (RWP) on the VMEWT write chain queue and signals the VMEWT activity to execute the request. The NSCRW write activity then waits on a push queue for the write operation to be completed or to time-out.

4. VMEWT receives the write request and controls the low-speed channel to transfer the data, in multiples of 512 words or less at a time. An MIOP Local Memory double buffering mechanism is used to write data to the low-speed channel concurrently with the transferring of the next sector of output data from the Buffer Memory buffers to the MIOP Local Memory.

5. VMEWT either completes the write or aborts it on an error and sets the status of the write for NSCRW. VMEWT then reactivates NSCRW with the information on the write request it just processed.

6. NSCRW deallocates the Buffer Memory buffers used in the request, returns the ending status in the N-packet to the mainframe, releases the RWP packet, and returns to step 1.

13.4 FLOW DESCRIPTION FOR SCP PROTOCOL

The following list describes the logon sequence for a station ID.

1. The driver activity (VMERD or NSCIO) calls NSCID to locate the Read/Write Packet (RWP) corresponding to an incoming message.

2. NSCID determines that the message is for the SCP protocol and checks that there is not an existing RWP on the driver activity's read chain for the station ID in the received message. NSCID confirms that this is a valid logon attempt.

3. NSCID allocates Local Memory for the Front-end Station Table (NT@) and for a RWP.

4. NSCID creates an SCPIO activity to handle communication between this station ID and COS/SCP.

5. SCPIO initializes and pushes itself, waiting to be activated by the driver activity when the read is complete.

6. After NSCID is notified that the SCPIO activity is in the wait state, it returns to the driver activity and to the address of the RWP it created for this station. NSCID returns execution to the driver activity.

7. The driver activity finishes reading any associated data and places information about the read operation in the RWP that NSCID returned to it. The driver activity then pops the SCPIO activity, signaling completion of the read.

8. SCPIO polls COS for a B-packet, which contains Cray Central Memory addresses for the incoming message and data, and then writes the message and any data to these addresses.

9. SCPIO waits for a response B-packet from COS. When it arrives, SCPIO reads the outgoing message and any data from the Cray Central Memory addresses specified in the response B-packet. Information about the outgoing message and possible data is put into the RWP.

10. SCPIO links the RWP to the end of the driver activity's write chain and pushes itself, waiting to be notified that the write is complete.

11. The driver activity takes the RWP off of its write chain, attempts the physical write (from information supplied in the RWP), and pops SCPIO with status of the attempt.

12. SCPIO queues another RWP on the driver activity's read chain and pushes itself, waiting for the driver activity to process a read for that RWP.

All future non-logon processing proceeds in a loop from step 12 to step 7, and then through step 12 again. For non-logon messages, NSCID returns to the driver activity, the address of RWP on the driver's read chain that has a station ID matching that of the incoming message. That RWP is the one that was placed on the read chain in step 12.

13.5 INTERRUPT HANDLING

The interrupt handling routine, IVME, resides in the Kernel. It places the ending channel address and ending channel status in the VMEbus driver table (input or output) relevant to the interrupted channel. The only error that the Kernel interrupt handler checks for is an output sequence error. The interrupt handler communicates with the driver read and write activities through the VR@KQ and VW@KQ queues, respectively.

14. PROGRAM LIBRARY AND MACROS

This section describes the I/O Subsystem (IOS) software program library (PL) and its structure. It also describes a set of macro instructions defined in $APTEXT. The process of adding an overlay, discussed at the end of this section, requires both the use of macros and modifications to the program library. The program library can be modified using the UPDATE program, described in the UPDATE Reference Manual, publication SR-0013.

14.1 PL STRUCTURE

The program library for the IOS software, IOPPL, has the following structure:

Deck	Contents
Common decks	
AT	System text, $APTEXT
eof	Deck containing *WEOF directive
K	Kernel
AMAP	Configuration overlay
Overlay decks	
TAPELOAD	Peripheral Expander tape deadstart program
DISKLOAD	Peripheral Expander disk deadstart program
DUMP	Dump program
eof	
CAL overlays	
eof	

The configuration AMAP overlay must be the first overlay.

The location of an overlay relative to others is based on the following considerations:

- Overlays are grouped by function (for instance, the station function).
- Within a group, overlays are alphabetized.

14.1.1 COMMON DECK STRUCTURE

The first and last statements in a common deck are comments indicating the scope of the common deck. Following the initial comment and preceding the final comment are list statements controlling the APML listing output. For example, the common deck DEB has the following structure:

```
|Location |Result     |Operand
|         |           |
|*        |Start of common deck DEB
|         |LIST       |OFF,NXRF
|DEB      |LIST       |ON,XRF
|         |  .        |
|         |  .        |
|         |  .        |
|DEB      |LIST       |  *
|DEB      |LIST       |  *
|*        |End of common deck DEB
```

14.1.2 ADDING AN OVERLAY

Adding an overlay to the IOS software requires the following:

- Inserting a deck containing the overlay into IOPPL. The location of the deck on the program library is dictated by considerations discussed in this section under the PL Structure heading.

- Defining the overlay number in deck OVLNUM. Overlay numbers have the form O$*name; name* is the overlay and overlay deck name. The overlay numbers in OVLNUM reflect the order of the decks on IOPPL, although it is not required that they do so.

The following example adds an overlay after the MULTIPLY overlay, assuming the overlay number for MULTIPLY is defined on line OVLNUM.157. The overlay expects two parameters. The overlay is in the grouping for overlays with the identifier $KOVL. (For descriptions of the OVERLAY and REGDEFS macros presented in the following example, see subsection 14.2.5, Overlay and Register Definition Macros.)

```
|Location |Result      |Operand              |Comment
|         |            |                     |
|*ID NEWDK,DC=OVLNUM   |                     |
|*I OVLNUM.157         |                     |
|NEWOVL   |OVNM        |                     |New overlay
|         |            |                     |
|*DK NEWOVL            |                     |
|         |LIST        |OFF,NXRF             |
|         |LISTOP      |(NEWOVL,$KOVL)       |
|         |OVERLAY     |NEWOVL               |Description of overlay
|         |  .         |                     |function and parameters
|         |  .         |                     |
|         |  .         |                     |
|         |REGDEFS     |,(PAR1,PAR2),(REG1,REG1,REG3)
|         |  .         |                     |Body of overlay
|         |  .         |                     |
|         |  .         |                     |
|         |END         |                     |
|*MOVEDK NEWOVL:MULTIPLY                     |
```

To obtain a listing of the overlay, specify the following on the appropriate APML statement:

```
LIST=NEWOVL
```

The overlay listing is included in the listing when specifying the following:

```
LIST=$KOVL
```

The groupings of overlays have the following identifiers that specify the group to be listed:

Overlay	Description
$BMX	Block multiplexer channel driver overlays
$CONC	Concentrator overlays
$COVL	CAL overlays
$UNICOS	UNICOS support overlays
$DIAG	Diagnostic and test overlays
$DSK	Disk driving overlays
$EXP	Expander device driving overlays
$FEI	FEI logical path overlays
$FILE	Minieditor and file system overlays
$HSX	HSX channel overlays
$INTER	Interactive station overlays
$KOVL	Kernel overlays
$NSC	NSC concentrator overlays
$SDMP	System dump and restart overlays

Overlay	Description
$STAT	Station overlays
$TAPE	On-line tape overlays
$TAPEIO	On-line tape overlays (I/O portion)
$TAPERR	On-line tape overlays (error recovery portion)
$UCHN	User Channel shell overlays
$VME	VMEbus interface overlays

14.2 MACROS

A set of macros is defined in the system text file, $APTEXT, for use in the IOS software.

Macros have been defined for sets of widely used functions, including the following:

- Exit stack access
- Execution control
- Table access
- Overlay and register definition

If used consistently, the macros make the IOS code more reliable, readable, and maintainable.

A macro may require a subset of the scratch registers %T1 through %T6. For overlays, these registers are allocated by the REGDEFS definition.

In general, unless otherwise specified, the operands for macros in this section can be the following:

- Constants
- Registers specified by the *R!* format
- Registers specified by the *(symbol)* format

The *(symbol)* format will often require an additional set of parentheses around the *(symbol)*, such as *((symbol))*. The extra outer set is necessary because parameters may be grouped with parentheses that are later stripped off by the macro processor.

Additional macros are defined in $APTEXT and the individual overlays. Table 14-1 summarizes the macros in alphabetical order. The Kernel service requests are also in this list; refer to subsection 2.9.3, I/O Operations, for more information on these requests.

Table 14-1. Summary of Macros

Name	Type	Function
A130OI	Kernel service request	Performs front-end I/O on an NSC channel
ADDRESS	Data access macro	Generates parcel address of a field
ALERT	Kernel service request	Creates an activity in another IOP
ASLEEP	Kernel service request	Suspends activity until an AWAKE message is received
AWAKE	Kernel service request	Activates an activity in another IOP
CALL	Kernel service request	Calls another overlay to perform a function
CLEAR	Memory macro	Zeros out area of Local Memory
COPY	Memory macro	Copies data from one memory location to another
CREATE	Kernel service request	Creates a new activity in the system
EDECR	Exit stack macro	Decrements exit stack pointer E by 1
EGET	Exit stack macro	Reads exit stack pointer E
EINCR	Exit stack macro	Increments exit stack pointer E by 1
EPUT	Exit stack macro	Sets exit stack pointer E
EXSGET	Exit stack macro	Reads exit stack address (E)
EXSPUT	Exit stack macro	Stores exit stack address (E)
FIELD	Data definition macro	Generates a set of symbols that define data field characteristics

Table 14-1. Summary of Macros (continued)

Name	Type	Function
FIND	Kernel service request	Returns the Buffer Memory address and length of an overlay
FLDADD	Data access macro	Adds to the contents of a field
FLDSUB	Data access macro	Subtracts from the contents of a field
FLUSH	Kernel service request	Releases all overlays in Local Memory
GET	Data access macro	Fetches contents of a field
GIVEUP	Kernel service request	Reschedules activity
GOTO	Kernel service request	Calls another overlay but does not save return information; does not return to caller of this overlay.
$GOTO	Execution control macro	Allows branching based on index value
$IF $ELSEIF $ELSE $ENDIF	Execution control macro	These associated macros allow for conditional execution, loop control, and indexed branching
ISFIELD	Data definition macro	Allows a new field to assume characteristics of the previous field
LOAD	Data access macro	Uses TABLE and FIELD definitions to generate tables at assembly time
MSG	Kernel service request	Sends message to Kernel console

Table 14-1. Summary of Macros (continued)

Name	Type	Function
MSGR	Kernel service request	Sends message to Kernel console and waits for operator response
OUTCALL	Kernel service request	Calls (through the CREATE function) an overlay in another IOP
OUTPUT	Kernel service request	Sends message to controlled CRT
OVERLAY	Overlay and register definition macro	Defines start of new overlay
PAUSE	Kernel service request	Suspends activity for specified tenths of a second
POP	Kernel service request	Reactivates pushed activity
$PUNTIF	Execution control macro	Provides conditional branch to Kernel halt routine
PUSH	Kernel service request	Deactivates activity until popped
PUT	Data access macro	Sets the contents of a field in a table
RECEIVE	Kernel service request	Waits for a character to be entered from a CRT controlled by USURP overlay
REGDEFS	Overlay and register definition macro	Defines overlay registers
REGISTER	Overlay and register definition macro	Associates names with particular operand register
RESPOND	Kernel service request	Sends message response to an activity in another IOP

Table 14-1. Summary of Macros (continued)

Name	Type	Function
RETREG	Overlay and register definition macro	Provides mechanism for returning value to a calling routine
RETURN	Kernel service request	Returns to caller of the overlay; if none, terminates activity.
RGET	Data access macro	Fetches the contents of a field
RPUT	Data access macro	Sets the contents of a field
RSTORE	Data access macro	Stores the contents of a field
TABLE	Data definition macro	Generates parameters for constructing tables
TERM	Kernel service request	Terminates activity
TPUSH	Kernel service request	Pushes activity until popped or time expires
$UNTIL $ENDTIL	Execution control macro	These associated macros provide loop control facility

14.2.1 EXIT STACK MACROS

The exit stack macros access the exit stack pointer (E) and the exit stack address (E). The macros altering the stack pointer or address provide the delay required to ensure that the Program Exit Stack channel transfer is complete. (See the Program Exit Stack channel description in the IOS hardware reference manual for your site.)

Interrupts must be disabled when executing the exit stack macros.

14.2.1.1 EGET macro

The EGET macro reads the exit stack pointer (E).

Format:

Location	Result	Operand
	EGET	*dest*

dest Register or memory location to receive the exit stack pointer value

The A register also receives the exit stack pointer value. EGET references the symbol @EGET. The cross-reference readily identifies exit stack pointer reads.

14.2.1.2 EPUT macro

The EPUT macro sets the exit stack pointer (E).

Format:

Location	Result	Operand
	EPUT	*source*

source Register or memory location containing the new exit stack pointer value

The A register receives the new exit pointer value. EPUT references the symbol @EPUT.

14.2.1.3 EINCR macro

The EINCR macro increments the exit stack pointer (E) by 1.

Format:

Location	Result	Operand
	EINCR	

The A register receives the new exit pointer value. EINCR references the symbol @EPUT.

14.2.1.4 EDECR macro

The EDECR macro decrements the exit stack pointer (E) by 1.

Format:

Location	Result	Operand
	EDECR	

The A register receives the new exit stack pointer value. EDECR references the symbol @EPUT.

14.2.1.5 EXSGET macro

The EXSGET macro reads the exit stack address (E).

Format:

Location	Result	Operand
	EXSGET	*dest*

dest Register or memory location to receive the exit stack address

The A register also receives the exit stack address. EXSGET references the symbol @EXSGET.

14.2.1.6 EXSPUT macro

The EXSPUT macro stores the exit stack address.

Format:

Location	Result	Operand
	EXSPUT	*source*

source Register or memory location for storing the exit stack address

EXSPUT references the symbol @EXSPUT.

Example:

The following example illustrates the use of the exit stack macros.

```
|Location |Result      |Operand       |Comment
|1        |10          |20            |35
|         |            |              |
|* Subroutine to save exit stack entries (entries 1 through n-1)
|* Enter:  R=SAVEXST   |              |
|* Inputs: R!REG0 holds Local Memory address at which to begin storing
|SAVEXST  |*           |              |
|         |EGET        |A             |. Get depth of exit stack
|         |R!REG1 = A - 1             |. Don't count this subroutine call
|         |R!REG2 = R!REG0 + R!REG1   |. Starting address for stack save
|         |EPUT        |0             |. Clear stack pointer
|         |$UNTIL (R!REG0 = R!REG2)   |. All entries saved?
|         |EINCR       |              |. Increment stack pointer
|         |EXSGET      |A             |. Get stack entry
|         |(R!REG0) = A               |. Save exit stack absolute address
|         |R!REG0 = R!REG0 + 1        |. Next storage location
|         |$ENDTIL     |              |
|         |EINCR       |              |. Get back to current subr call
|         |EXIT        |              |
```

14.2.2 EXECUTION CONTROL MACROS

The execution control macros define the path of execution through the code. Macros for conditional execution, loop control, and indexed branching are available. The macros reduce the number of explicitly defined labels required for a particular code sequence and improve the readability of the code.

14.2.2.1 $IF macro

The $IF macro, and the associated $ELSEIF, $ELSE, and $ENDIF macros, allow conditional execution of code in the manner of an if-then-else construct.

Format:

Location	Result	Operand	Comment
1	10	20	35
	$IF	*condition*	
	.		Execute if condition
	.		is true.
	.		
	$ELSEIF	*condition*	
	.		Execute if preceding
	.		conditions are false and
	.		this condition is true.
	$ELSE		
	.		Execute if all of the
	.		above conditions
	.		are false.
	$ENDIF		

condition

An expression taking one of the following forms where $cond_i$ is any expression that is a legal condition on an APML instruction:

($cond_1$)	True if $cond_1$ is true
($cond_1$), OR, ($cond_2$)	True if either $cond_1$ or $cond_2$ is true
($cond_1$), AND, ($cond_2$)	True if both $cond_1$ and $cond_2$ are true

The $ELSEIF and $ELSE macros are optional. Only one $ELSE macro is allowed. Multiple $ELSEIF macros are allowed, but they must precede the $ELSE macro.

$IF structures can be nested up to 10 levels.

Example:

Location	Result	Operand
	$IF	(P1>P2)
		R!MAX=P1
	$ELSE	
		R!MAX=P2
	$ENDIF	

14.2.2.2 $UNTIL macro

The $UNTIL macro and the associated $ENDTIL macro provide a loop control facility.

Format:

Location	Result	Operand	Comment
1	10	20	35
	$UNTIL	*condition*	
	.		Execute if condition is
	.		false.
	.		
	$ENDTIL		

condition Loop termination condition. It is an expression taking one of the following forms, where $cond_i$ is any expression that is a legal condition on an APML instruction:

$(cond_1)$	True if $cond_1$ is true
$(cond_1)$, OR, $(cond_2)$	True if either $cond_1$ or $cond_2$ is true
$(cond_1)$, AND, $(cond_2)$	True if both $cond_1$ and $cond_2$ are true

The code between the $UNTIL and $ENDTIL is executed until the condition becomes true, at which time execution resumes immediately following the $ENDTIL.

$UNTIL loops can be nested up to 10 levels.

Normally, the body of the loop is modifying some resource used in the loop condition.

Example:

Location	Result	Operand	Comment
1	10	20	35
	B=0		
	$UNTIL	(B=O'50)	For channels 0 to 47
	IOB: 0		Clear channel done and busy.
	B=B+1		
	$ENDTIL		

14.2.2.3 $GOTO macro

The $GOTO macro allows branching based on an index value.

Format:

Location	Result	Operand
	$GOTO	($label_0$,$label_1$, ... ,$label_n$),
		index[,BASE=*basereg*]

$label_i$ Branch address, relative to the base address in *basereg*. A null label indicates a fall-through condition.

index Index used to select a branch address (0 through *n*). An index greater than *n* is a fall-through condition.

BASE=*basereg*
Base to be added to the branch address. If not specified, the contents of operand register %B, which is the overlay base address register, are used.

Indexing begins with 0, not 1.

Example:

```
|Location |Result      |Operand          |Comment
|1        |10          |20               |35
|         |            |                 |
|*        |Branch per function code in FC
|*        |            |                 |
|         |$GOTO       |(,OPEN,,CLOSE),FC
|*        |            |                 |
|*        |Here, if FC = 0, FC = 2, or FC > 3
|         |  .         |                 |
|         |  .         |                 |
|         |  .         |                 |
|OPEN     |  *         |                 |.FC = 1
|         |  .         |                 |
|         |  .         |                 |
|         |  .         |                 |
|CLOSE    |  *         |                 |.FC = 3
```

14.2.2.4 $PUNTIF macro

The $PUNTIF macro provides a conditional branch to the Kernel halt routine.

Format:

Location	Result	Operand
	$PUNTIF	*condition*,[CODE=*code*]

condition An expression taking one of the following forms, where $cond_i$ is any expression that is a legal condition on an APML instruction:

($cond_1$)	True if $cond_1$ is true
($cond_1$), OR, ($cond_2$)	True if either $cond_1$ or $cond_2$ is true
($cond_1$), AND, ($cond_2$)	True if both $cond_1$ and $cond_2$ are true

code Error code to be output with the Kernel halt message. The code must be less than 1000_8. If no code is specified, 0 is used. The Kernel error codes are listed in subsection 15.5, LISTP Overlay.

Example 1:

```
|Location |Result      |Operand        |Comment
|1        |10          |20             |35
|         |            |               |
|         |IOR:10      |               |. Get interrupting channel number
|         |IA = A      |               |. Save the number
|         |IC = EITB + IA              |. Index into the interrupt handler
|         |            |               |. address jump table by channel no
|         |IC = (IC)   |               |. Get interrupt handler address
|         |            |               |. for this channel
|         |P = IC, A # 0               |. Jump to this routine if it
|         |            |               |. exists
|         |$PUNTIF     |CODE=PT$UXINT  |. PUNT unconditionally if the
|         |            |               |. routine does not exist for this
|         |            |               |. unexpected channel interrupt
```

Example 2:

```
|Location |Result     |Operand       |Comment
|1        |10         |20            |35
|         |           |              |
|* High-speed channel I/O routine    |
|* Moves data between Local Memory and target memory
|* Target memory must be the SSD or Cray Central Memory
|* Maximum data length is 512 64-bit words
|* Inputs:  R!TR4 = length in words  |
|*          R!TR0 = target memory type
|* Now, check inputs ...             |
|          |$PUNTIF    |(R!TR0 # F$CMEM) AND (R!TR0 # FS$SSD),
|          |           |CODE=PT$BADTM|. PUNT if target memory is not
|          |           |             |. SSD or Cray Central Memory
|          |$PUNTIF    |(R!TR4 > 1000), CODE=PT$IOLEN
|          |           |             |. PUNT if requested length is
|          |           |             |. too long
```

14.2.3 DATA DEFINITION MACROS

The data definition macros define tables and fields to be used later by the access macros.

14.2.3.1 FIELD macro

The FIELD macro generates a set of symbols that define the characteristics of a data field.

Format:

```
|Location |Result     |Operand
|         |           |
|name     |FIELD      |parcel,sbit,width[,L=length]
```

name Field name; 1 to 6 characters.

parcel Relative parcel within the table or table entry in which the field resides. If *, *name*@P is not defined.

* Suppresses the definition of *name@p*

$ Indicates that the current parcel (the last explicitly given parcel, or the last parcel number generated by a FIELD or TABLE macro), is to be used. This format is useful when new fields are added to the middle of a large table.

+ Indicates that the current parcel +1 is to be used

sbit Leftmost bit number of the field within the parcel. Bit 0 is the leftmost bit. * suppresses the *name@S* and *name@N* definitions; *width* must be omitted if * is used.

width Number of bits in the field. This parameter is not required if *sbit* is *. If not specified, 16_{10} is assumed and *name*@M and *name*@X are not defined. The sum of *sbit* and *width* must not exceed 16_{10}; that is, a field must not cross a parcel boundary.

L=*length* Optional parcel length of the field. If specified, *length* is added to *parcel* to form a new current parcel number for the next FIELD macro.

Parameters *parcel, sbit, width,* and *length* are assumed to be decimal unless otherwise indicated.

The FIELD macro generates a subset of the following set of symbols:

Symbol	Value
name@P	Parcel offset
name@S	Starting bit number
name@N	Bit width
name@M	Mask for the field, right-justified
name@X	Mask, the ones complement of *name*@M

14.2.3.2 ISFIELD macro

The ISFIELD macro allows a newly defined field definition to assume the same characteristics as a previously defined field.

Format:

Location	Result	Operand
symbol	ISFIELD	*previous*

symbol Name of a new field

previous Name of a previously defined field

Example:

```
|Location |Result      |Operand          |Comment
|1        |10          |20               |35
|         |            |                 |
|*        |DAL Description               |
|DA@      |TABLE       |LH=8,LE=24       |. DALs are 32 parcels
|         |            |                 |. 24 parcel entry is the
|         |            |                 |. Cray to IOS communication
|         |            |                 |. packet
|*        |Header Description            |
|DA@LNK   |FIELD       |0                |. Link cell
|DA@IFC   |FIELD       |1                |. IOP function code
|         |.           |                 |
|         |.           |                 |
|         |.           |                 |
|DA@MES   |FIELD       |5                |. Message for accumulator
|         |            |                 |. channel
|DA@QUE   |FIELD       |5,0,4            |. Message queue (includes IOP#)
|DA@NUM   |FIELD       |5,2,2            |. IOP number
|DS@IND   |FIELD       |5,4,12           |. MOS message in
|DA@SID   |FIELD       |9                |. Source ID
|         |.           |                 |
|         |.           |                 |
|         |.           |                 |
|DA@LOC   |FIELD       |31               |. Local Buffer address
|         |            |                 |
|         |            |                 |
|*        |Disk Error Packet Description
|DE@      |TABLE       |LH=8,LE=24       |
|*        |FIELD       |0                |
|DE@DID   |ISFIELD     |DA@DID           |. Destination ID
|DE@SID   |ISFIELD     |DA@SID           |. Source ID
|         |.           |                 |
|         |.           |                 |
|         |.           |                 |
|DE@CV    |FIELD       |24,*             |. Beginning of correction vector
|         |            |                 |. buffer within the packet (does
|         |            |                 |. not generate the definition
|         |            |                 |. DE@CV@P
```

14.2.3.3 TABLE macro

The TABLE macro initializes the parcel designator for subsequent FIELD macros and generates a set of symbols that define the characteristics of a table.

Format:

Location	Result	Operand
name	TABLE	[LH=*header*][,LE=*entry*][,NE=*number*] [,SZ=*size*]

name Name of the table

LH=*header* Length of the table header

LE=*entry* Length of the table entry

NE=*number* Number of entries in the table

SZ=*size* Size of the complete table

The TABLE macro generates a subset of the following set of symbols, which correspond to the parameters in the call:

Symbol	Description
name@LH	Header length
name@LE	Entry length
name@NE	Number of entries
name@SZ	Size of the table; if SZ=* is specified on the call, this is equal to (*name*@LE * *name*@NE) + *name*@LH.

14.2.4 DATA ACCESS MACROS

The data access macros manipulate tables and fields defined by the data definition macros.

14.2.4.1 ADDRESS macro

The ADDRESS macro calculates the parcel address of a field.

Format:

Location	Result	Operand
	ADDRESS	*result,field,base*

result Register or memory location to receive the address

field Field name

base Table or entry base address; *base* can be an operand register, memory address, or the contents of a memory location.

14.2.4.2 GET macro

The GET macro fetches the contents of a field.

Format:

Location	Result	Operand
	GET	*dest,field,base*

dest Register or memory location to receive the field contents

field Field name

base Base address

The A register also receives the contents of a field.

14.2.4.3 LOAD macro

The LOAD macro creates a table, a field, or both in place at assembly time from the TABLE and FIELD definitions.

Format:

Location	Result	Operand
[*type*]	LOAD	[*name*][,*value*]

type One of the following types of load to perform:

Type	Function
TABLE	Creates a table from the symbols specified in a previously defined TABLE macro. If *name* is not supplied, no table is created. If *name* is supplied and the symbol *name*@SZ is defined, a complete table is created. If *name* is supplied but *name*@SZ is not defined, a table is created using the *name*@LH, *name*@LE, and *name*@NE symbols or by using *value*. (*value* contains the number of entries when used with TABLE.) The table is zero-filled.
HEADER	Generates a header for a table with the table name of *name*
ENTRY	Generates one table entry with the table name of *name*
FIELD	Creates one field with the name of *name* and the contents of *value*; this is the default.

name Name assigned in a previously defined TABLE macro

value Either the contents of the field when used with the FIELD type or the number of entries when used with the TABLE type

Example:

```
|Location |Result    |Operand          |Comment
|1        |10        |20               |35
|         |          |                 |
|CPW@     |TABLE     |LE=4             |
|CPW@FL   |FIELD     |0,0,8            |
|CPW@CM   |FIELD     |0,8,8            |
|CPW@DA   |FIELD     |1,*              |
|CPW@BU   |FIELD     |2,*              |
|CPW@BL   |FIELD     |3,*              |
|         |          |                 |
|*        |          |                 |
|*        |Generate a predefined CPW   |
|*        |          |                 |
|         |          |                 |
|TABLE    |LOAD      |                 |
|         |LOAD      |CPW@FL,CPW$CC    |.Chaining
|         |LOAD      |CPW@CM,CM$SNS    |.Sense
|         |LOAD      |CPW@DA,BA        |.Data address
|         |LOAD      |CPW@BL,24        |.Byte length
|ENTRY    |LOAD      |CPW@             |.Complete entry
|         |LOAD      |CPW@CM,X'7       |.Rewind
|ENTRY    |LOAD      |CPW@             |.Complete entry
```

14.2.4.4 PUT macro

The PUT macro sets the contents of a field. The PUT macro cannot be used to store a constant; see the STORE macro for that function.

Format:

Location	Result	Operand
	PUT	*source,field,base*

source Register or memory location containing value to be stored

field Field name

base Base address

The source field is updated with the logical product of the value and the field mask.

14.2.4.5 STORE macro

The STORE macro stores a constant in a field.

Format:

Location	Result	Operand
	STORE	*constant,field,base*

constant Value to be stored in the field

field Field name

base Base address

14.2.4.6 RGET macro

The RGET macro fetches the contents of a field. The parcel containing the field is contained in an operand register or a particular memory location and, unlike the GET macro, is not loaded from a table.

Format:

Location	Result	Operand
	RGET	*dest,field,parcel*

dest Register or memory location to receive the field contents

field Field name

parcel Operand register or memory location containing the parcel holding the field

The A register also receives the contents of a field.

14.2.4.7 RPUT macro

The RPUT macro sets the contents of a field. The parcel containing the field is contained in an operand register or a particular memory location and, unlike the PUT macro, is not stored into a table. The RPUT macro cannot be used to store a constant; see the RSTORE macro for that function.

Format:

Location	Result	Operand
	RPUT	*source,field,parcel*

source Register or memory location containing value to be stored. It is updated with the logical product of the value and the field mask.

field Field name

parcel Operand register or memory location containing the parcel holding the field

14.2.4.8 RSTORE macro

The RSTORE macro stores a constant in a field. The parcel containing the field is contained in an operand register or a particular memory location and, unlike the STORE macro, is not stored into a table.

Format:

```
|Location |Result     |Operand
|         |           |
|         |RSTORE     |constant,field,parcel
```

constant Value to be stored in a field

field Field name

parcel Operand register or memory location containing the parcel holding the field

Example:

```
|Location |Result     |Operand          |Comment
|1        |10         |20               |35
|         |           |                 |
|DH@JOB   |FIELD      |4,*              |Job name
|DH@DC    |FIELD      |10               |Disposition code
|DH@ED    |FIELD      |47,4,12          |Edition number
|         | .         |                 |
|         | .         |                 |
|         | .         |                 |
|         |           |                 |
|*        |Operand register SEG contains table address.
|*        |Operand register DC contains disposition code.
|*        |           |                 |
|         |PUT        |R!DC,DH@DC,R!SEG
|         |STORE      |0,DH@ED,R!SEG    |
|         |ADDRESS    |R!%W1,DH@JOB,R!SEG
|         |           |                 |Enter job name at this address
|         | .         |                 |
|         | .         |                 |
|         | .         |                 |
|IO@STA   |FIELD      |8                |Stream state
|IOS@SB   |FIELD      |*,0,4            |Stream control byte
|IOS@IO   |FIELD      |*,4,1            |Send/receive
|         |           |                 |Message flag
|         | .         |                 |
|         | .         |                 |
|         | .         |                 |
|*        |           |                 |
|*        |Operand register STATE contains the table entry address
|*        |           |GET              |R!%W1,IO@STA,R!STATE
|         |RGET       |R!SCB,IOS@SB,R!%W1
|         |RGET       |R!FLAG,IOS@IO,R!%W1
```

14.2.4.9 FLDADD macro

The FLDADD macro adds to the contents of a field.

Format:

```
|Location |Result    |Operand
|         |          |
|         |FLDADD    |number,field,base
```

number Number or location of the number to be added to the contents of the field

field Field name

base Base address

The A register receives the updated contents of the field.

Example:

```
|Location |Result    |Operand          |Comment
|1        |10        |20               |35
|         |          |                 |
|* R!DQ has the executable DAL address|
|* R!MDAL has the master DAL address  |
|         |GET       |R!%W1,DA@WBH,R!MDAL
|         |          |                 |. Next write-behind sector
|         |GET       |R!%W2,DA@SEQ,R!DQ
|         |          |                 |. Sector just moved
|         |$IF       |(R!%W1 = R!%W2)  |. If same sector
|         |FLADD     |1,DA@WBH,R!MDAL  |. Advance the write-behind
|         |$ENDIF    |                 |
```

14.2.4.10 FLDSUB macro

The FLDSUB macro subtracts from the contents of a field.

Format:

```
|Location |Result    |Operand
|         |          |
|         |FLDSUB    |number,field,base
```

number Number or location of the number to be subtracted from the contents of the field

field Field name

base Base address

The A register receives the updated contents of the field.

14.2.5 OVERLAY AND REGISTER DEFINITION MACROS

The overlay and register definition macros define the environment but do not generate code.

14.2.5.1 OVERLAY macro

The OVERLAY macro identifies the beginning of an overlay and generates an overlay header containing the overlay name, number, type, and input parameter register information. By default, this macro expects decimal values for parameters.

Format:

Location	Result	Operand
	OVERLAY	*name*[,TYPE=*type*][,EXTN=*extn*]

name Overlay name

type Overlay type; the default is IOP.

Type	Function
IOP	The overlay is executable through a Kernel CREATE, CALL, or GOTO function. %B is defined as the base register. Input parameter registers must be defined within the overlay through the REGDEFS macro.
DATA	The overlay is not executable under Kernel control.
CAL	The overlay is coded in CAL.

extn This option is for use in the overlay OVLNUM in which the overlay numbers are defined. Specify NO if the overlay number is not to be declared as an external.

The overlay number O$*name* must be defined before the OVERLAY statement is encountered. O$$OVL is equated to the overlay number.

OVERLAY sets the numeric base to octal.

14.2.5.2 REGDEFS macro

The REGDEFS macro defines the operand registers for an overlay. The macro generates the symbol %P, which is the register number of the next available register to follow %W5. It also generates the symbol %NP, which is the number of parameter registers.

Format:

Location	Result	Operand
start	REGDEFS	(*global*),(*param*),(*local*)

start Starting register number; the default is the $APTEXT parameter %GBLREG.

global List of names to associate with the global registers. The global registers are saved and restored on a task-by-task basis, not on an overlay-by-overlay basis. The registers are automatically passed to a called overlay. Any modifications performed by the called overlay are reflected on return to the calling overlay. The maximum number of global registers is defined by the $APTEXT parameter %GBLNUM. The beginning global register number is defined by the $APTEXT parameter %GBLREG.

param List of names to associate with the operand registers containing input parameters. Parameters are passed to the overlay in the registers named by *param*. The number of parameters specified on the call do not need to match the number of parameter registers defined by *param*. The symbol %P is equated to the first entry in the list (even if null) and %NP is set to the number of entries in the list. These symbols are used by the OVERLAY macro to generate the overlay header. Within the overlay and during Kernel service requests, the parameter registers are treated like the local registers.

local List of names associated with operand registers used within the overlay. When the overlay is entered, the contents of the local registers are undefined. The local registers (and parameter registers) are preserved across a Kernel service request (including a call to another overlay) unless the register is designated on the request as a receptacle for returned information.

In any of the lists, a register can be assigned without an associated name by specifying * as the parameter. Null arguments in the list are ignored.

In addition to the explicitly listed registers, the REGDEFS macro automatically assigns names to other operand registers as follows:

Name	Use
%S1 through %S5	Scratch registers used for temporary storage by APML. The overlay code should never directly reference these registers.
%T1 through %T6	Scratch registers used by macros. When writing macros, the following procedures should be noted: • Macro registers are generally not saved across Kernel service requests. • Nested macros must use disjoint sets of scratch registers.
%W1 through %W5	Scratch registers that can be used explicitly within the overlay. These registers are not, however, preserved across Kernel service requests.

14.2.5.3 REGISTER macro

The REGISTER macro associates names with particular operand registers. The macro generates the symbol $REGORG, which is the value of the next available register.

Format:

Location	Result	Operand
first	REGISTER	(*list*)

first Operand register to associate with the first name in list. If *first* is not specified, operand register allocation proceeds where the previous REGISTER definition left off. It must be used the first time the REGISTER macro is encountered in a program module.

list List of names to associate with the successive operand registers. A null argument in the list does not cause a register to be assigned. A register can be assigned without an associated name by specifying an asterisk (*) as the parameter.

Example:

Location	Result	Operand	Comment
1	10	20	35
200	REGISTER	(DA,DB,DC,DD)	Disk interrupt answering
	REGISTER	(DE,DF)	Disk interrupt answering

The following operand registers are associated with the names indicated (register numbers are in octal):

Operand Register	Name
200	DA
201	DB
202	DC
203	DD
204	DE
205	DF

14.2.5.4 RETREG macro

The RETREG macro provides a mechanism for returning a value to a calling routine. The value is stored in the SMOD of the caller. When a RETURN is executed, the caller's registers and any returned parameters are loaded.

Format:

```
|Location |Result     |Operand
|         |           |
|         |RETREG     |value,dest
```

value Constant or register containing the value to be returned to the caller

dest Caller's register designator:

A = A register
B = B register
C = Carry bit
offset = Register containing an offset designating one of the caller's operand registers. Normally, this offset is provided as an RO=*reg* parameter on the CALL statement (see the CALL function description in section 2).

Example:

```
|Location |Result     |Operand          |Comment
|1        |10         |20               |35
|         |OVERLAY    |BMXCON           |
|* BMXCON is called to vary the status of a block mux component
|* Inputs: component type               |
|*         channel number               |
|*         control unit address         |
|*         device ordinal               |
|*         up/down indicator            |
|         |REGDEFS    |,(TYPE,CHN,CTU,DVN,STATE),(CHT,RCD,CUT,DVT,CRW)
|         |           |                 |. Registers 430 through 434
|         |           |                 |. are parameter registers;
|         |           |                 |. registers 435 through 441
|         |           |                 |. are local registers.
|         |           |                 |. %P=430, %NP=5
|         |.          |                 |
|         |.          |                 |
|         |.          |                 |
|* Exit point         |                 |
|* A device status must be returned     |
|* 0 means operational device, nonzero means nonoperational device
|         |RETREG     |R!%W5,A          |. Send status back to the field
|         |           |                 |. that holds the accumulator
|         |           |                 |. value in the previous SMOD;
|         |           |                 |. that is, the SMOD of the
|         |           |                 |. caller.
```

14.2.6 MEMORY MACROS

Memory macros clear Local Memory areas or copy data from one memory location to another.

14.2.6.1 CLEAR macro

The CLEAR macro is used to zero out an area of Local Memory.

Format:

Location	Result	Operand
	CLEAR	START=*exp1*,COUNT=*exp2*[,BLANKS=NO|YES]

exp1 An expression that evaluates to the starting address for clearing

exp2 An expression that evaluates to the number of parcels to clear

BLANKS NO is the default and causes zeros to be used to clear the area. YES can be specified to cause ASCII blanks to fill the area.

14.2.6.2 COPY macro

The COPY macro copies data from one memory location to another.

Format:

Location	Result	Operand
	COPY	*from,to,length*

from Source address

to Destination address

length Number of data parcels

Example:

```
|Location |Result     |Operand              |Comment
|1        |10         |20                   |35
|         |           |                     |
|* Send a message to the console            |
|         |R!LEN = 40 |                     |. Message is 40 parcels
|         |           |                     |. or 80 ASCII characters
|         |CLEAR      |START=R!BUF,COUNT=R!LEN
|         |           |                     |. Clear buffer to zeros
|         |COPY       |R!MES+R!%B,R!BUF,R!LEN
|         |           |                     |. Copy message data from the
|         |.          |                     |. overlay to buffer
|         |MSG        |R!BUF                |. Output the message
```

15. DEBUGGING TOOLS

This section describes tools for debugging and maintaining I/O Subsystem (IOS) software. See the COS Operational Procedures Reference Manual, publication SM-0043, for station debugging commands.

The history trace and the debugger provide on-line access to the inner workings of the code. Also, the debugger permits the modification of registers and memory on an on-line basis.

The PATCH overlay allows a location within an overlay or within the Kernel to be displayed and modified.

The LISTO overlay prints the names and numbers of all overlays defined in the Kernel. It includes information such as the Buffer Memory address and length for each overlay.

15.1 SUMMARY UTILITY

The SUMMARY utility provides a display (see figure 15-1) of the dynamic characteristics for a specified I/O Processor (IOP). The display is available on any station console. The command format and display field descriptions follow.

SUMMARY is meant to be used as a tool to examine the characteristics of an IOP while it is running. It can also provide information on how modifications to the system affect the overall behavior.

Format:

```
 _____________________________
|                             |
| SUMMARY,iop (defaults to 0) |
|_____________________________|
```

iop Number of the IOP for which the display is active

The interval for which percentages and counts are displayed is determined by the refresh rate of the station console on which the display is being viewed.

NOTE

The SUMMARY display uses a significant proportion of IOP resources when active. It should not be left running during normal operation.

IOS SUMMARY DISPLAY IOP: 0 %IDLE: 99 %SYS: 8 FRAME: 0

OVERLAYS						INTERRUPTS				KERNEL CALLS			
NAME	%Tm	NUM	Tus	K-C	INT	CH	%Tm	NUM	Tus	NAME	NUM	Tus	INT
CLOCK	1	6	17	0	0	4	1	6059	9	PUSH	1	10	0
CRTDEM	4	39	15	0	3	17	0	43	27	POP	2	12	0
DEVICE	0	2	28	2	0	40	0	1	9	DLAY	1	5	0
SUMDAY	1	2	32	3	0	41	0	32	9	TPSH	43	14	0
TSTASH	0	2	22	2	0					OUTP	6	12	0
XPRINT	25	82	50	82	10					RCVE	6	12	0
XPRNTA	2	7	40	7	1					MGET	1	13	0
CONSL	7	9	122	21	1					MOSR	29	17	0
DISP01	0	2	27	2	0					MOSW	7	17	0
COMM04	1	2	27	3	0					CALL	63	15	0
AMPEX	3	11	43	11	0					RTRN	61	9	0
CLI	2	2	150	2	1								

Figure 15-1. The SUMMARY Display

Fields displayed in figure 15-1:

%IDLE — Percent of time during the last interval spent in the idle loop

%SYS — Percent of nonidle time spent in activity swapping, including SMOD maintenance and overlay loading

FRAME — Current frame number. (+ advances the frame and a - reverses the frame.) The maximum frame number is 7.

15.1.1 OVERLAYS

Only overlays that were active over the last interval are displayed as follows:

Heading	Description
NAME	First 6 ASCII characters of the overlay name
%Tm	Percent of activity time (nonidle and non-sys) spent in the displayed overlay
NUM	Number of times over the last interval that the displayed overlay was activated
Tus	Maximum interval, in microseconds, for which the overlay was in control

NOTE

An interval is terminated by either a voluntary termination by the overlay or an interrupt.

K-C	Number of service requests made by the overlay
INT	Number of interrupts that occurred while the overlay was in control

15.1.2 INTERRUPTS

Only channels that interrupted over the interval are displayed. Their descriptions are as follows:

Heading	Description
CH	Channel number (in octal) of the interrupting channel
%Tm	Percent of activity time (nonidle) for which the interrupt handling for the displayed channel was in control
NUM	Number of interrupts received from the displayed channel
Tus	Maximum time spent in the interrupt handler over the last interval

15.1.3 KERNEL CALLS

Only kernel calls (service requests) that were active over the last interval are displayed. Their descriptions are as follows:

Heading	Description
NAME	A 4-character ASCII description of the service request
NUM	Number of calls made to the displayed service request over the last interval
Tus	Maximum interval (in microseconds) for which the displayed service request was in control
INT	Number of interrupts that has occurred while the displayed service request was in control

15.2 HISTORY TRACE

The IOS history trace stores in a buffer pertinent data relating to selected events (for instance, a Kernel service call). This data can be examined at any time. The TLOC pointer in the Kernel table area points to the beginning of the Local Memory trace buffer. TPTRL points to the location within the Local Memory Buffer where the next trace entry will be tabled. Each IOP configured maintains its own history trace buffer. The buffer includes a small Local Memory buffer that, when filled, is dumped to a large circular buffer in Buffer Memory.

Trace buffers can be examined on-line using the TRACE call or off-line with a dump command. Subsection 15.2.1 discusses the use of TRACE on-line to examine trace buffers and subsection 15.2.2 discusses the COS and UNICOS dump commands used to examine trace buffers off-line. Subsection 15.2.3 provides tables of the event codes, subcodes, and parameters used in traces and dumps.

15.2.1 EXAMINING TRACE BUFFERS ON-LINE

The TRACE command gives the user control of the trace mechanism. Event codes can be selectively enabled and disabled, and a formatted listing of the current trace buffer can be obtained. A command in the following format, entered at any Kernel console, allows the selection of individual events to be recorded or bypassed, depending on whether ON or OFF is selected:

```
TRACE ON|OFF event[/subcode]
```

ON|OFF	Enables or disables the trace for the event specified
event	Octal code associated with the event recorded (see table 15-1)
/subcode	Code used with events TF$CHN(4) (channel interrupts) and TF$FCT(5) (Kernel functions). The subcode for a channel interrupt is the octal channel number itself. The subcode for a Kernel function is its function code (see table 15-2 for a listing of the function codes that are supported). If the subcode is omitted, the associated subevents are all affected by the call. The subcode is entered as parameter 1 on trace output.

Only one event code can be specified in a single TRACE command. However, the following command affects all events:

```
TRACE ON|OFF ALL
```

The following command specifies whether to dump the local trace buffer to the circular buffer in Buffer Memory:

```
TRACE ON|OFF MOS
```

The following command causes either the Local or Buffer Memory (MOS) buffer to be formatted and output to the expander printer (entries are formatted from most recent to least recent):

```
TRACE DUMP LOCAL|MOS
```

Each trace entry is 8 parcels long and is formatted as follows:

event	*time*	*overlay*	par_1	par_2	par_3	par_4	par_5

event Octal code associated with the recorded event (see table 15-1). The octal code appears in a new or unformatted octal dump. When using TRACE or dump formatting commands however, the octal code is replaced by a mnemonic.

time Low-order 16 bits of real-time clock at time of recording (in millisecond increments)

overlay Number of the overlay whose Local Memory base address was in the Kernel operand register %B at the time of the recording. Number 177777_8 represents the Kernel.

par_i Selected parameter recorded with event. Table 15-2 lists the parameters recorded for each event.

Figure 15-2 shows a sample page of formatted output from the history trace operating in the Master I/O Processor (MIOP).

IOP-Ø TRACE BUFFER

EVENT	RTC-L	OVL-#	PAR-1	PAR-2	PAR-3	PAR-4	PAR-5
TASK	110076	000601	046744	000006	000601	101630	000007
OVL-LD	110076	000601	000601	046744	000014	000001	054514
MEM-IO	110076	000602	000001	000001	017756	046744	000204
K-FNCT	110076	000602	CALL	066726	000601	012054	006430
K-CALL	110076	000602	CALL	000704	066640	101630	000502
TASK	110076	000602	055230	002470	000602	101630	000007
INTRPT	110076	177777	000034	014422	000000	000072	112415
K-FNCT	110076	000602	NSCIO	101530	000004	101534	000101
K-CALL	110076	000602	NSCIO	002470	066640	101630	000502
TASK	110076	000602	055230	003077	000602	101630	000007
INTRPT	110076	177777	000034	014422	000000	000072	111761
K-FNCT	110076	000602	NSCIO	101540	000040	101534	000045
K-CALL	110076	000602	NSCIO	003077	066640	101630	000502
TASK	110075	000602	055230	000562	000602	101630	000007
OVL-LD	110075	000602	000602	055230	000013	000001	054513
MEM-IO	110075	177777	000001	000001	020162	055230	000716
MEM-IO	110075	177777	000001	000002	125600	066640	000020
MEM-IO	110075	177777	000004	000002	121600	066640	000011
INTRPT	110075	177777	000034	014422	000000	000072	107247
K-FNCT	110067	000440	PUSH	077324	077324	076006	000000
K-CALL	110067	000440	TPUSH	000070	066640	077324	000445
TASK	110067	000440	045410	001210	000440	077324	000007
K-CALL	110067	000436	RETURN	000073	066671	077324	000460
TASK	110067	000436	060730	002731	000436	077324	000007
K-FNCT	110067	000436	MPUT	000001	122000	000001	000000
K-CALL	110067	000436	MPUT	002731	066671	077324	000460
TASK	110067	000436	060730	002174	000436	077324	000007
K-CALL	110067	000551	RETURN	001045	066735	077324	000457
TASK	110067	000551	052674	001257	000551	077324	000007
K-CALL	110067	000425	RETURN	000771	067001	077324	000455
TASK	110067	000425	033614	001072	000425	077324	000007
MEM-IO	110067	000425	000002	000001	036503	075414	000021
K-CALL	110067	000425	MOSR	001072	067001	077324	000455
TASK	110067	000425	033614	000006	000425	077324	000007
K-FNCT	110067	000551	CALL	067001	000425	011174	006430
K-CALL	110067	000551	CALL	001257	066735	077324	000457
TASK	110067	000551	052674	001257	000551	077324	000007
K-CALL	110067	000425	RETURN	000701	067001	077324	000455
TASK	110067	000425	033614	001072	000425	077324	000007
MEM-IO	110067	000425	000002	000001	036462	075414	000021
K-CALL	110067	000425	MOSR	001072	067001	077324	000455
TASK	110067	000425	033614	000006	000425	077324	000007
K-FNCT	110067	000551	CALL	067001	000425	011174	006430

1813

Figure 15-2. History Trace Sample Output

15.2.2 EXAMINING TRACE BUFFERS OFF-LINE

Trace buffers can be examined off-line by issuing a dump command. The command used depends on the operating system running on the mainframe.

The COS dump command for examining trace buffers is FDUMP. See Operational Aids Reference Manual, publication SM-0044, for a complete description of FDUMP.

Figure 15-3 shows an example of the output from FDUMP.

```
DMEM,TYPE=IOPO,FWA=0,LWA=177777,R.                          FDUMP 1.16      11/16/87    15:48:10    PAGE   125
     DXIR TYPE=IOPO                                         SYSDUMP         05/07/87    10:33:06

EVENT   RTC-1   OVL    PAR-1          PAR-2          PAR-3          PAR-4          PAR-5

TASK    141711  CRIDEM BASE-ADD 043214 P-REL  000006 OVL.#  000023 ACT.ADD 072760 MOS-LOCL 000000
INTRPT  141711  KERNEL CHANNEL# 000041 P.ADDR 014417 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 174741
TASK    141710  CRIDEM BASE-ADD 043214 P-REL  000006 OVL.#  000023 ACT.ADD 072760 MOS-LOCL 000000
INTRPT  141710  CDEM   CHANNEL# 000041 P.ADDR 041642 BASE/OVL 041554 IDLE-HI 000006 IDLE-LO 163067
INTRPT  141710  CDEM   CHANNEL# 000011 P.ADDR 041642 BASE/OVL 041554 IDLE-HI 000006 IDLE-LO 163067
TASK    141710  CDEM   BASE-ADD 041554 P-REL  000066 OVL.#  000016 ACT.ADD 073020 MOS-LOCL 000000
K-CALL  141710  CDEM   FUNCT    RELDAL P-REL  000066 SMOD.ADD 073044 ACT.ADD 073020 B.REG.   000456
A-IO-A  141710  CDEM   CHANNEL# 000011 MESSAGE 140046 CODE.#  000001 IOP/CHAN 016026 R/W.CODE 000400
MEM-IO  141710  CDEM   FNCT.CDE 000003 MEM BITS 000000 MEM BITS 011707 MEM-LOC 070344 LNGTH-WD 000010
TASK    141710  CDEM   BASE-ADD 041554 P-REL  000006 OVL.#  000016 ACT.ADD 073020 MOS-LOCL 000000
INTRPT  141710  KERNEL CHANNEL# 000020 P.ADDR 014416 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 163067
I-HAND  141710  KERNEL CHAN# IN 000020 STATUS 000000 DAL ADDR 070344 CHANNEL# 070404 DESTIN  000101
INTRPT  141710  KERNEL CHANNEL# 000021 P.ADDR 014412 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 162246
ICOM    141710  ICOM   CODE     000001 DAL.ADDR 070304 MESSAGE 140026 IOP/CH# 002026 R/W CODE 000001
TASK    141710  ICOM   BASE-ADD 056314 P-REL  000156 OVL.#  000132 ACT.ADD 073160 MOS-LOCL 000000
MEM-IO  141710  ICOM   FNCT.CDE 000002 MEM BITS 000000 MEM BITS 011507 MEM-LOC 070304 LNGTH-WD 000010
K-CALL  141710  ICOM   FUNCT    MOSR   P-REL  000156 SMOD.ADD 073204 ACT.ADD 073160 B.REG.   000471
TASK    141710  ICOM   BASE-ADD 056314 P-REL  000052 OVL.#  000132 ACT.ADD 073160 MOS-LOCL 000000
K-CALL  141710  ICOM   FUNCT    GETDAL P-REL  000052 SMOD.ADD 073204 ACT.ADD 073160 B.REG.   000471
TASK    141710  ICOM   BASE-ADD 056314 P-REL  000006 OVL.#  000132 ACT.ADD 073160 MOS-LOCL 000000
INTRPT  141710  KERNEL CHANNEL# 000010 P.ADDR 014411 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 162230
I-HAND  141710  KERNEL IOP.CHN# 000010 MESSAGE 140026 INTRRPTS 100045 DALS-LOC 000057 MOS-LOC  000000
TASK    141710  CRIDEM BASE-ADD 043214 P-REL  000006 OVL.#  000023 ACT.ADD 072760 MOS-LOCL 000000
INTRPT  141710  KERNEL CHANNEL# 000041 P.ADDR 014411 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 153000
INTRPT  141707  CDEM   CHANNEL# 000011 P.ADDR 041642 BASE/OVL 041554 IDLE-HI 000006 IDLE-LO 146404
TASK    141707  CDEM   BASE-ADD 041554 P-REL  000066 OVL.#  000016 ACT.ADD 073020 MOS-LOCL 000000
K-CALL  141707  CDEM   FUNCT    RELDAL P-REL  000066 SMOD.ADD 073044 ACT.ADD 073020 B.REG.   000456
A-IO-A  141707  CDEM   CHANNEL# 000011 MESSAGE 140045 CODE.#  000001 IOP/CHAN 016024 R/W.CODE 000400
MEM-IO  141707  CDEM   FNCT.CDE 000003 MEM BITS 000000 MEM BITS 011677 MEM-LOC 066704 LNGTH-WD 000010
TASK    141707  CDEM   BASE-ADD 041554 P-REL  000006 OVL.#  000016 ACT.ADD 073020 MOS-LOCL 000000
INTRPT  141707  KERNEL CHANNEL# 000020 P.ADDR 014416 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 146404
I-HAND  141707  KERNEL CHAN# IN 000020 STATUS 000000 DAL ADDR 066704 CHANNEL# 066744 DESTIN  000101
INTRPT  141707  KERNEL CHANNEL# 000021 P.ADDR 014416 BASE/OVL 000000 IDLE-HI 000006 IDLE-LO 145554
ICOM    141707  ICOM   CODE     000001 DAL.ADDR 066744 MESSAGE 140036 IOP/CH# 002024 R/W CODE 000001
TASK    141707  ICOM   BASE-ADD 056314 P-REL  000156 OVL.#  000132 ACT.ADD 073160 MOS-LOCL 000000
MEM-IO  141707  ICOM   FNCT.CDE 000002 MEM BITS 000000 MEM BITS 011607 MEM-LOC 066244 LNGTH-WD 000010

                                                                                                    1517
```

Figure 15-3. FDUMP Sample Output

The UNICOS dump command for examining trace buffers is **fdmp**(1M). See the UNICOS System Administrator's Guide, publication SR-2022, for a complete description of **fdmp**.

Figure 15-4 shows an example of the output from **fdmp**.

```
$ /etc/fdmp -m iop0 -t -T 10 core/core.xxxx

/etc/fdmp: iop0 memory from file core/core.xxxx          Dump date: 07/16/88
13:12:44                              Page 1
Sysdump: iop-3 halt 000

Event   Clock   Overly          Ent-1   Ent-2   Ent-3   Ent-4   Ent-5

Chanio  015037  177777          000012  177000  160252  000112  000006
Task    015036  Bcom            045364  000275  000151  101244  000006
Memio   015036  Bcom            000002  000000  013020  074114  000010
K-call  015036  Bcom            Mosr    000275  101270  101244  000473
Task    015036  Bcom            045364  000275  000151  101244  000006
Memio   015036  Bcom            000002  000000  013010  072214  000010
K-call  015036  Bcom            Mosr    000275  101270  101244  000473
Intrpt  015036  Bcom            000012  045372  045364  000062  071726
Chanio  015036  Bcom            000012  040126  160251  000112  000006
Task    015036  Bcom            045364  000006  000151  101244  000006

****************** Dump completed
```

1518

Figure 15-4. **fdmp** Sample Output

15.2.3 TRACE EVENT CODES, SUBCODES, AND PARAMETERS

Table 15-1 contains trace event codes, and table 15-2 adds subcodes and parameters to the Trace Event Codes in table 15-1.

Table 15-1. Trace Event Codes

Event	Code	Description
INTRPT† Intrpt††	TF$INT (1)	Exit from common interrupt handler (ICHK)
K-CALL† K-call††	TF$CALL (2)	Entrance to Kernel function processor (SERVICE)
TASK† Task††	TF$TSK (3)	Exit from activity dispatching (ELDP)

† Mnemonic used by on-line TRACE and off-line COS FDUMP command.
†† Mnemonic used by off-line UNICOS **fdmp**.

Table 15-1. Trace Event Codes (continued)

Event	Code	Description
I-HAND† Chanio††	TF$CHN (4)	Individual interrupt handlers (IIAP, IEXP, and ICRI)
K-FNCT† K-func††	TF$FCT (5)	Individual Kernel function processor
SEEK† D2seek††	TF$SEK (6)	Disk seek routine (SEEK)
DK-IO† D2stio††	TF$DSK (7)	Disk read/write processor (DDRF)
DK-ERR† D2err††	TF$DSKER (10)	Entry to disk error recovery (ERRECK overlay)
OVL-LD† Ldovly††	TF$OLAY (12)	Overlay loading (EIAK)
A-TO-A† A-Asnd††	TF$ATA (13)	Send messages to other IOPs (EMSGIOP)
ACOM† Acomsg††	TF$ACOM (14)	Receive messages from other IOPs (MEMX in ACOM overlay)
DK-MM† D2micr††	TF$DKMM (15)	Disk error recovery retries; usually micro positioning attempts (ERRECK overlay).
DK-LOG† D2elog††	TF$DKLOG (16)	Disk error logging; signals the end of recovery processing for a disk error (REPORT overlay).
CRTOUT† Ochar††	TF$CHAR (17)	CRT output character (CRTDEM)
BMX0† Bmx-in††	TF$BMX0 (20)	BMXDEM - Entrance
BMX1† Bmx-io††	TF$BMX1 (21)	BMXDEM - Start I/O
BMX2† Bmx-ad††	TF$BMX2 (22)	BMXDEM - Advance data

† Mnemonic used by on-line TRACE and off-line COS FDUMP command.
†† Mnemonic used by off-line UNICOS **fdmp**.

Table 15-1. Trace Event Codes (continued)

Event	Code	Description
BMX3† Bmx-si††	TF$BMX3 (23)	BMXSIO - Entrance
BMX4† Bmx-ap††	TF$BMX4 (24)	BMXSIO - Path assignment
TEX0† Bcomsg††	TF$TEX0 (30)	BCOM3 - CPU request
TEX1† Bcm-rs††	TF$TEX1 (31)	BCOM3 - BIOP response
TEX2† Bfm-in††	TF$TEX2 (32)	BUFMAN - Entrance
TEX3† Bfm-dn††	TF$TEX3 (33)	BUFMAN - Exit
TEX4† Tpi-rs††	TF$TEX4 (34)	TAPEIO - BMX response
TEX5† Tdm-in††	TF$TEX5 (35)	TDEM - Entrance
TEX6† Tdm-dn††	TF$TEX6 (36)	TDEM - Exit
TEX7† Td0-in††	TF$TEX7 (37)	TEDM1 - Entrance
TEX8† Td0-dn††	TF$TEX8 (40)	TDEM1 - Exit
TEX9† 000041††	TF$TEX9 (41)	TAPEIO - CPU response
TEXA† 000042††	TF$TEXA (42)	TAPEIO - DSC Buffer Descriptor Entry read
BYPIO† 000046††	TF$BYPIO (46)	Issue I/O between Buffer Memory and a target memory by using the high-speed bypass channel.

† Mnemonic used by on-line TRACE and off-line COS FDUMP command.
†† Mnemonic used by off-line UNICOS **fdmp**.

Table 15-1. Trace Event Codes (continued)

Event	Code	Description
TMOUT† Evtout††	TF$TOUT (47)	Clock-event time-out
MEM-IO† Memio††	TF$COMIO (50)	Issue I/O to Buffer Memory and a Target Memory
ICOM† Icomsg††	TF$ICOM (51)	Receive messages from other IOPs (ICOM)
D4-SEK† D4seek††	TF$D4SK (52)	DCU-5 type disk seek routine (D4DEM)
D4-HED† D4head††	TF$D4HD (53)	DCU-5 type disk head select routine (D4DEM)
D4-IO† D4stio††	TF$D4IO (54)	DCU-5 type disk read/write routine (D4DEM and DD49)
D4-ERR† D4err††	TF$D4ER (56)	DCU-5 type disk error recovery retry (D3ERR and D4ERR)
UCH0† Sh-ent††	TF$UCH0 (60)	UCSHL - Entrance
UCH1† Sh-exi††	TF$UCH1 (61)	UCSHL - Exit
UCH2† Sh-sio††	TF$UCH2 (62)	UCRD/UCWRT - Entrance
UCH3† Sh-iod††	TF$UCH3 (63)	UCRD/UCWRT - Exit
UCH4† Sh-req††	TF$UCH4 (64)	Driver call - Entrance
UCH5† Sh-res††	TF$UCH5 (65)	Driver call - Exit
SCPIO† 000070††	TF$SCP1 (70)	Concentrator status

† Mnemonic used by on-line TRACE and off-line COS FDUMP command.
†† Mnemonic used by off-line UNICOS **fdmp**.

Table 15-1. Trace Event Codes (continued)

Event	Code	Description
NSCRW† 000071††	TF$LPH1 (71)	NSC logical path driver request status
FEIW/FEIR† 000072††	TF$LPH@ (72)	FEI logical path driver request status

† Mnemonic used by on-line TRACE and off-line COS FDUMP command.
†† Mnemonic used by off-line UNICOS **fdmp**.

Table 15-2. Trace Event Parameters

Event	Parameters
INTRPT or Intrpt TF$INT (1) Exit from common interrupt handler	P1 = Channel number P2 = P address P3 = Base address of overlay P4 = Idle time (IOP) high-order bits P5 = Idle time (IOP) low-order bits
K-CALL or K-call TF$CALL (2) Entrance to Kernel function processor	P1 = Function P2 = Address relative to the base address P3 = SMOD address P4 = Activity Descriptor address P5 = B register
TASK or Task TF$TSK (3) Exit from activity dispatching	P1 = Base address P2 = Address relative to the base address P3 = Overlay number P4 = Activity Descriptor address P5 = Number of available local buffers
I-HAND or Chanio TF$CHN (4) Input A-A channel interrupt handlers	P1 = IOP input channel number P2 = Message (see table 2-4) P3 = Interrupt count P4 = Number of available local DALS P5 = Number of available local buffers
I-HAND or Chanio TF$CHN (4) Expander channel interrupt	P1 = Expander channel number P2 = Status P3 = Device status P4 = Address of first entry on PUSH queue P5 = Address of last entry on PUSH queue

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
I-HAND or Chanio TF$CHN (4) Input channel from mainframe interrupt	P1 = Input channel number P2 = Status P3 = Address of DAL P4 = Current channel address P5 = Destination ID (DA@DID)
K-FNCT or K-func TF$FCT (5) Individual Kernel function processor (PUSH) Deactivates activity until popped	P1 = Function 1 = PUSH P2 = First entry on linked list P3 = Last entry on linked list P4 = Queue address P5 = 0
K-FNCT or K-func (POP) Reactivates pushed activity	P1 = Function 2 = POP P2 = Linked activity P3 = Queue address P4 = 0 P5 = 0
K-FNCT or K-func (ALERT) Creates an activity in another IOP	P1 = Function 15 = ALERT P2 = IOP number (0 through 3) P3 = Number of created overlay P4 = DAL address high-order bits P5 = DAL address low-order bits
K-FNCT or K-func (AWAKE) Activates an activity in another IOP	P1 = Function 16 = AWAKE P2 = IOP number (0 through 3) P3 = Popcell address P4 = DAL address high-order bits P5 = DAL address low-order bits
K-FNCT or K-func (RESPOND) Sends message response to activity in another IOP	P1 = Function 17 = RESPOND P2 = Address of DAL P3 = Response P4 = Destination IOP (0 through 3) P5 = 0
K-FNCT or K-func (NSCIO) Initiate I/O to A130 NSC device	P1 = Function 24 = NSCIO P2 = Input address P3 = Input length P4 = Output address P5 = A130 function or output length

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
K-FNCT or K-func (GETMEM) Allocates Local Memory in multiples of 4	P1 = Function 30 = GETMEM P2 = Length in parcels P3 = Address P4 = 0 P5 = 0
K-FNCT or K-func (RELMEM) Releases memory to free pool	P1 = Function 31 = RELMEM P2 = Length in parcels P3 = Address P4 = 0 P5 = 0
K-FNCT or K-func (SEND) Initiates output to mainframe through 6 Mbyte channel	P1 = Function 34 = SEND P2 = Address of DAL P3 = 0 P4 = 0 P5 = 0
K-FNCT or K-func (MGET) Gets Buffer Memory from buffer pool	P1 = Function 35 = MGET P2 = Buffer Memory address high-order bits P3 = Buffer Memory address low-order bits P4 = Number of buffers allocated P5 = 0
K-FNCT or K-func (MPUT) Return Buffer Memory to free pool	P1 = Function 36 = MPUT P2 = Buffer Memory address high-order bits P3 = Buffer Memory address low-order bits P4 = Number of buffers to return P5 = 0
K-FNCT or K-func (POLL) OUT-C1 - Initiates output to main-frame through 6 Mbyte channel	P1 = Function 44 = POLL, also tagged OUT-C1 some places P2 = Address of DAL P3 = 0 P4 = 0 P5 = 0
K-FNCT or K-func (CALL) Calls another overlay to perform a function	P1 = Function 50 = CALL P2 = SMOD address P3 = Overlay number P4 = Address of entry in overlay table P5 = Registers saved. Bits 0 through 6 give the number saved; bits 7 through 15 identify first to be saved.

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
K-FNCT or K-func (CREATE) Creates a new activity in the system	P1 = Function 55 = CREATE P2 = Overlay number P3 = Activity Descriptor address P4 = SMOD address P5 = SMOD size
SEEK or D2seek TF$SEK (6) Disk seek routine	P1 = Channel number P2 = Old cylinder number P3 = New cylinder number P4 = IOP idle time high-order bits P5 = IOP idle time low-order bits
DK-IO or D2stio TF$DSK (7) Disk read/write processor	P1 = Function code in first 7 bits, channel number in last 9 P2 = Cylinder number in first 11 bits, head in last 5 P3 = Sector number P4 = Local Buffer address P5 = Caller's return address
DK-ERR or D2err TF$DSKER (10) Disk error handler	P1 = Channel number P2 = Error type P3 = Channel activity flags P4 = Address of current executable DAL P5 = Disk Control Block address
OVL-LD or Ldovly TF$OLAY (12) Overlay loading	P1 = Overlay number P2 = Base address P3 = Number of overlays in Local Memory P4 = High-order bits of total number of loads for all overlays P5 = Low-order bits of total number of loads for all overlays
A-TO-A or A-Asnd TF$ATA (13) Send messages to other IOPs	P1 = Channel number P2 = Message (see table 2-4) P3 = Internal function code (see following) P4 = IOP number in first 7 bits, channel number in last 9 P5 = Read/write function code (DAF$)

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
A-TO-A or A-Asnd (continued)	CODE 1 = Initiate disk I/O 2 = Release DAL space 3 = Transfer data from Central Memory to Buffer Memory (BIOP receives) 4 = Transfer data from Buffer Memory to Central Memory (BIOP receives) 5 = Send status to COS (MIOP receives) 6 = Transfer from mainframe to Buffer Memory complete (DIOP receives) 7 = Transfer from Buffer Memory to mainframe complete (DIOP receives)
ACOM or Acomsg TF$ACOM (14) Receives messages from other IOPs (in overlay ACOM)	P1 = Code (see code meanings immediately preceding) P2 = DAL address P3 = Message (see table 2-4) P4 = First 4 bits contain the device type; last 3 bits contain the IOP number. P5 = Cylinder number in first 11 bits, head number in last 5
DK-MM or D2micr TF$DKMM (15) Disk ERRECK Disk retries	P1 = Channel number P2 = Cylinder number first 11 bits, head number error recovery in last 5 P3 = Sector number P4 = Type of error P5 = Retry count
DK-LOG or D2elog TF$DKLOG (16) Disk error logging	P1 = Channel number P2 = Type of error P3 = Recovered/unrecovered status P4 = Disk Control Block address
CRTOUT or Ochar TF$CHAR (17) CRT output character	P1 = Channel number P2 = Character (octal representation of ASCII) P3 = Address of local buffer P4 = Activity Descriptor address P5 = 0
BMXO or Bmx-in TF$BMX0 (20) BMXDEM - ENTRANCE	P1 = Device ordinal P2 = Channel number P3 = Channel input tags P4 = Device status P5 = Sequence code (KIC$)

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
BMX1 or Bmx-io TF$BMX1 (21) BMXDEM - START I/O	P1 = Current CPW address P2 = Device command P3 = CPW flags P4 = Device path P5 = CPB flags (CPB@FL)
BMX2 or Bmx-ad TF$BMX2 (22) BMXDEM - ADVANCE DATA	P1 = Device ordinal P2 = Current CPW address P3 = CPB flags (CPB@FL) P4 = Data buffer address P5 = Data byte count
BMX3 or Bmx-si TF$BMX3 (23) BMXSIO - ENTRANCE	P1 = Device ordinal P2 = Request code (RQ$) P3 = CPB address P4 = 0 P5 = 0
BMX4 or Bmx-ap TF$BMX4 (24) BMXSIO - PATH ASSIGNMENT	P1 = Device table address P2 = Channel table address P3 = Control unit table address P4 = Channel number P5 = Device path
TEX0 or Bcomsg TF$TEX0 (30) BCOM3 - CPU REQUEST	P1 = Device ordinal P2 = Function (FC$) P3 = Requested block count P4 = Requested sector count P5 = Dataset description flags
TEX1 or Bcm-rs TF$TEX1 (31) BCOM3 - BIOP RESPONSE	P1 = Device ordinal P2 = Transferred block count P3 = Transferred sector count P4 = Valid Buffer Memory sector/block count P5 = Status (TQ@STS)
TEX2 or Bfm-in TF$TEX2 (32) BUFMAN - ENTRANCE	P1 = Device ordinal P2 = Pointer to top of DSC list P3 = Pointer to bottom of DSC list P4 = XIOP pointer into DSC list P5 = BIOP pointer into DSC list

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
TEX3 or Bfm-dn TF$TEX3 (33) BUFMAN - EXIT	P1 = Device ordinal P2 = Pointer to top of DSC list P3 = Pointer to bottom of DSC list P4 = XIOP pointer into DSC list P5 = BIOP pointer into DSC list
TEX4 or Tpi-rs TF$TEX4 (34) TAPEIO - BMX RESPONSE	P1 = Device ordinal P2 = Operation status (OS$) P3 = CPB error flags (CPB@EF) P4 = Block done count P5 = Blocks left in command chain
TEX5 or Tdm-in TF$TEX5 (35) TDEM - ENTRANCE	P1 = Device ordinal P2 = Operation status (OS$) P3 = CPB error flags (CPB@EF) P4 = Pointer into DSC list for current segment P5 = CPB address
TEX6 or Tdm-dn TF$TEX6 (36) TDEM - EXIT	P1 = Device ordinal P2 = Bytes transferred to or from Buffer Memory for current command (high-order bits) P3 = Bytes transferred to or from Buffer Memory for current command (low-order bits) P4 = Current CPU address P5 = CPB flags and DSC Buffer Descriptor Entry status. Flags are bits 0-11. The status is valid only on write operations.
TEX7 or Td0-in TF$TEX7 (37) TDEM1 - ENTRANCE	P1 = Device ordinal P2 = Requested sector count P3 = BIOP pointer into DSC list P4 = Pointer into current sector for next word P5 = Pointer into current word for next byte
TEX8 or Td0-dn TF$TEX8 (40) TDEM1 - EXIT	P1 = Transferred block count P2 = Transferred sector count P3 = BIOP pointer into DSC list P4 = Pointer into current sector for next word P5 = Pointer into current word for next byte

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
TEX9 or 000041 TF$TEX9 (41) TAPEIO - CPU RESPONSE	P1 = Status (TQ@STS) P2 = Blocks/sectors in Buffer Memory P3 = Maximum block size; high-order bits P4 = Maximum block size; low-order bits P5 = 0
TEXA or 000042 TF$TEXA (42) - DSCREAD	P1 = Pointer into DSC list P2 = DSC Buffer Descriptor Entry status P3 = Contiguous sector count P4 = Record length; high-order bits P5 = Record length; low-order bits
BYPIO or 000046 TF$BYPIO (46) Issue I/O between Buffer Memory and a target memory on bypass channel	P1 = Function code in upper 2 bits, target memory type in next 2 bits, word length in last 12 bits P2 = High-order bits of target memory address P3 = Low-order bits of target memory address P4 = High-order bits of Buffer Memory address P5 = Low-order bits of Buffer Memory address
TMOUT or Evtout TF$TOUT (47) Clock-event time-out	P1 = Timer entry link address P2 = Time-out routine address P3 = 0 P4 = 0 P5 = 0
MEM-IO or Memio TF$COMIO (50) Issue I/O between Local Memory and a Target Memory	P1 = Function code (see following) P2 = High-order bits of a target memory address P3 = Low-order bits of a target memory address P4 = Local Memory address P5 = Word length

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
MEM-IO or Memio (continued)	Function 1 = Buffer Memory read, wait until done 2 = Buffer Memory read, do not wait until done 3 = Buffer Memory write, wait until done 4 = Buffer Memory write, do not wait until done 5 = Central Memory read, wait until done 6 = Central Memory read, do not wait until done 7 = Central Memory write, wait until done 10 = Central Memory write, do not wait until done 11 = SSD Memory read, wait until done 12 = SSD Memory read, do not wait until done 13 = SSD Memory write, wait until done 14 = SSD Memory write, do not wait until done
ICOM or Icomsg TF$ICOM (51) Receive messages from other IOP for DCU-5 type disks (overlay ICOM)	P1 = Internal function code (IM$) P2 = DAL address P3 = Accumulator message (see table 2-4) P4 = IOP number in first 4 bits, unit number in next 3 bits, and channel in last 9 bits P5 = Read/write code (DAF$)
D4-SEK or D4seek TF$D4SK (52) DCU-5 seek routine (overlay D4DEM)	P1 = Channel P2 = Old cylinder P3 = New cylinder P4 = DCB address P5 = 0
D4-HED or D4head TF$D4HD (53) DCU-5 head select (overlay D4DEM)	P1 = Channel P2 = Old head group P3 = New head group P4 = DCB address P5 = 0
D4-IO or D4stio TF$D4IO (54) DCU-5 sector I/O (overlays DD49 and D4DEM)	P1 = Read/write code in first 10 bits, channel in last 6 bits P2 = Unit number in first 4 bits, cylinder number in last 12 bits P3 = Head group P4 = Sector P5 = Local Memory buffer

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
D4-ERR or D4err TF$D4ER (56) DCU-5 error retry	P1 = Unit number in first 4 bits, channel in last 12 bits P2 = Channel activity flags at time of error P3 = Error type (IE$) P4 = Current error status code (S$) P5 = Retry count
UCH0 or Sh-ent TF$UCH0 (60) UCSHL - ENTRANCE	P1 = Channel number P2 = User Channel Table address P3 = User Channel state P4 = F-packet request address P5 = Function code
UCH1 or Sh-exi TF$UCH1 (61) UCSHL - EXIT	P1 = Channel number P2 = User Channel Table address P3 = User Channel state P4 = F-packet request address P5 = Response status
UCH2 or Sh-sio TF$UCH2 (62) UCRD/UCWRT - ENTRANCE	P1 = Channel number P2 = High-order bits of Data Length 1 P3 = Low-order bits of Data Length 1 P4 = High-order bits of Data Length 2 P5 = Low-order bits of Data Length 2
UCH3 or Sh-iod TF$UCH3 (63) UCRD/UCWRT - EXIT	P1 = Channel number P2 = High-order bits of Transfer Length 1 P3 = Low-order bits of Transfer Length 1 P4 = High-order bits of Transfer Length 2 P5 = Low-order bits of Transfer Length 2
UCH4 or Sh-req TF$UCH4 (64) Driver call - ENTRANCE	P1 = Channel number P2 = User Channel Table address P3 = Command code P4 = Local buffer address P5 = I/O size in bytes
UCH5 or Sh-res TF$UCH5 (65) Driver call - EXIT	P1 = Channel number P2 = User Channel Table address P3 = Response code P4 = Local buffer address P5 = I/O size in bytes

Table 15-2. Trace Event Parameters (continued)

Event	Parameters
SCPIO or 000070 TF$SCP1 (70) Concentrator status	P1 = State of concentrator P2 = Front-end table address P3 = Packet address P4 = Error register P5 = Station ID
NSCRW or 000071 TF$LPH1 (71) Status of NSC logical path driver request	P1 = Logical path P2 = Function/subfunction code P3 = Status P4 = Request address P5 = Logical path table address
FEIW/FEIR or 000072 TF$LPH2 (72) Status of FEI logical path driver request	P1 = Channel number P2 = Address of request packet P3 = Status of input operation P4 = Front-end interface table address P5 = 0

15.3 DEBUGGER

The IOS debugging utility has the following capabilities:

- Sets up to four breakpoints
- Executes I/O instructions
- Examines and modifies the following:
  - Operand registers
  - Operand pointer (B register)
  - Accumulator (A register)
  - Carry bit (C)
  - Exit stack (E register)
  - Local Memory
  - Buffer Memory

The debugger is composed of two common decks assembled with the Kernel: DEBUGTOO and DEBUGGER. Deck DEBUGTOO contains resident routines and debugger data areas. Deck DEBUGGER contains the bulk of the debugger code. This code is relocatable and must be less than 511 words in length. The relocatable code is allocated to Local Memory in an IOP when the debugger is entered during initialization or when called through the DEBUG command. The Local Memory is released when the BUGOFF command is entered.

To enter the debugger during initialization perform the following steps:

1. When deadstarting the IOP, respond to the loader prompt message (FILE @MT0: or FILE @DK0:) with a CONTROL-D. The loader returns the message DEBUG and reissues the prompt message. Respond with the appropriate tape file number or disk file name.

2. The loader reads in the Kernel and passes control to it. In the MIOP, the debugger gains control. To stop in the debugger in another IOP during initialization, set the appropriate bit in the cell DEBUG before proceeding:

Bit	IOP
14	BIOP
13	IOP-2
12	IOP-3

If the debugger is entered during initialization, breakpoints can be set in both initialization and post-initialization code. Breakpoints set in the MIOP during initialization do not cause breakpoints to be set in the other IOPs.

NOTE

Breakpoints set in code executed prior to debugger initialization cause unpredictable results in the other IOPs.

After initialization, the DEBUG and BUGOFF commands control the debugger.

Enter the DEBUG command at the console attached to the IOP in which the debugger is to be used. The command loads the debugger into Local Memory (if it is not already loaded) and sets the debug base register to the Local Memory address. The debugger is then given control and accepts commands. Once the DEBUG command is entered, the debugger remains in Local Memory until the BUGOFF command is entered.

Format:

```
 _______
|       |
| DEBUG |
|_______|
```

The BUGOFF command clears all breakpoints, returns the Local Memory buffer allocated for the debugger, and sets the debug base register to 0. Before entering the BUGOFF command, enter the X command first (as described later in this section).

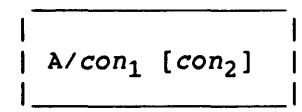
Format:



The following subsections describe the debug commands. The formats for these commands show a slash following the command if the response is displayed before a carriage return is entered. If there is no slash following the command, a carriage return must follow the command.

15.3.1 DISPLAY ACCUMULATOR COMMAND

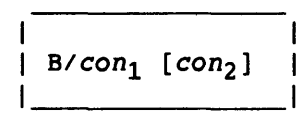
This command has the following format:



Type A to display the accumulator. The debugger responds with a slash and the current accumulator contents (con_1). The accumulator contents can be modified by immediately typing the new value (con_2) followed by a carriage return. If the accumulator is not to be changed, enter a carriage return only.

15.3.2 DISPLAY B REGISTER COMMAND

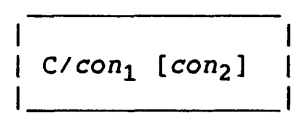
This command has the following format:



Type B to display the B register. The debugger responds with a slash and the current B register contents (con_1). The B register contents can be modified by typing the new value (con_2) followed by a carriage return. If the B register is not to be changed, enter a carriage return only.

15.3.3 DISPLAY CARRY REGISTER COMMAND

This command has the following format:



Type C to display the carry register. The debugger responds with a slash and the current carry register contents (con_1). The carry register contents can be modified by typing the new value (con_2) followed by a carriage return. If the carry register is not to be changed, enter a carriage return only.

15.3.4 DISPLAY CHANNEL STATUS COMMAND

This command has the following format:



The status of channel *n* is displayed.

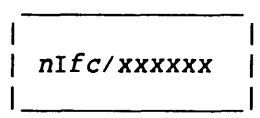
15.3.5 ISSUE A FUNCTION ON A CHANNEL COMMAND

This command has the following format:



The function *fc* is issued on channel *n* with the A register's initial value 0. The status of the channel is displayed after the function is issued.

Format:



The contents of the A register (*xxxxxx*) resulting from the function (*fc*) are displayed along with the status of the *n* channel done and busy flags. If a carriage return terminates this command, the operation is performed once. If a line feed terminates the command, the same operation is performed for each line feed entered.

15.3.6 DISPLAY EXIT STACK COMMAND

This command has the following format:

```
 ____________________
|                    |
| [n]E/con1 [con2]   |
|____________________|
```

Type E to display the E register, the exit stack pointer. Type the exit stack entry index (*n*) followed by E to display the contents of the exit stack at position *n* (default is 0). The debugger responds with a slash and the current register contents (con_1). The register contents may be modified by immediately typing the new value (con_2) followed by a carriage return. If the register is not to be changed, enter a carriage return only. Typing a line feed instead of a carriage return completes processing of the specified entry and displays the next exit stack entry and contents for possible modification.

15.3.7 DISPLAY OPERAND REGISTER COMMAND

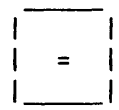
This command has the following format:

```
 ___________________
|                   |
| nR/con1 [con2]    |
|___________________|
```

Type the operand register number (*n*) followed by R to display the contents of operand register *n*. The debugger responds with a slash and the current operand register contents (con_1). The operand register contents can be modified by immediately typing the new value (con_2) followed by a carriage return. If the operand register is not to be changed, enter a carriage return only. Entering a line feed instead of a carriage return completes the processing of the specified register and then displays the next operand register and its contents for possible modification. Typing a slash instead of a carriage return completes processing of the specified register and then displays the contents of storage at the location specified by the register contents; storage can be modified by entering the new value followed by a carriage return. If storage is not to be changed, enter a carriage return only.

15.3.8 TOGGLE DISPLAY MODE COMMAND

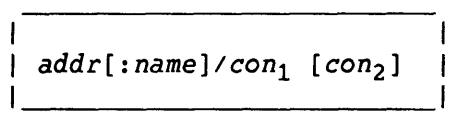
This command has the following format:



Type = to toggle the default display mode between absolute and overlay-relative. The default mode setting on each entry into the debugger is absolute. The overlay selected is the last one referenced either implicitly (by occurrence of a breakpoint) or explicitly (by setting a breakpoint in an overlay). Toggling to overlay-relative mode is not allowed if the overlay is not in Local Memory. The default display mode affects Local Memory and P register displays.

15.3.9 DISPLAY LOCAL MEMORY COMMAND

This command has the following format:

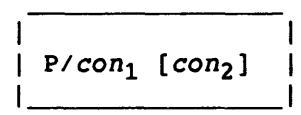


Type the address followed by a slash to display Local Memory. If *:name* is included, the contents of location *addr* of overlay *name* are displayed. The debugger responds with the current contents (con_1). The contents can be modified by immediately typing the new value (con_2) followed by a carriage return. If the contents are not to be changed, enter a carriage return only.

Typing a line feed instead of a carriage return completes the processing of the current Local Memory location and then displays the next address and contents for possible modification. Entering a slash instead of a carriage return completes the processing of the current location and then displays the contents of the location specified by the current location contents. Local Memory can be modified by entering the new value followed by a carriage return. If it is not to be changed, enter a carriage return only.

15.3.10 DISPLAY P REGISTER COMMAND

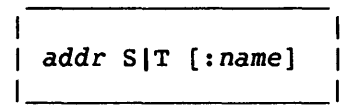
This command has the following format:



Type P to display the P register. The absolute P address or the overlay-relative P address is displayed. The debugger responds with a slash and the current P register contents (*con*$_1$). The P register contents can be modified immediately by typing the new value (*con*$_2$) followed by a carriage return. If the P register is not to be modified, enter a carriage return only.

15.3.11 SET SINGLE BREAKPOINT COMMAND

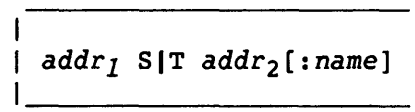
This command has the following format:



Type the breakpoint address followed by S or T to set a breakpoint. An S specifies that the breakpoint address is in noninterruptible code; the T specifies interruptible code. If *:name* is included, the breakpoint is set at location *addr* of overlay *name*. A maximum of four breakpoints can be active simultaneously. The first available breakpoint number is assigned to the breakpoint. A breakpoint cannot be set at address 0. A single breakpoint is cleared when the breakpoint is encountered.

15.3.12 SET DOUBLE BREAKPOINT COMMAND

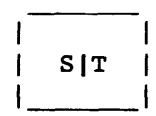
This command has the following format:



Type the primary breakpoint address ($addr_1$) followed by S or T, followed by the alternate breakpoint address ($addr_2$) to set a double breakpoint. An S specifies that the breakpoint address is in noninterruptible code; the T specifies interruptible code. If :*name* is included, the breakpoints are set in overlay *name*. A double breakpoint stops at the primary breakpoint when it is first encountered and stops each subsequent time that the primary breakpoint is encountered after encountering the alternate breakpoint. A double breakpoint is cleared only by the *n*D or D command.

15.3.13 DISPLAY BREAKPOINTS COMMAND

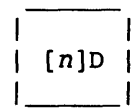
This command has the following format:



Type S or T to display all active breakpoints.

15.3.14 DELETE BREAKPOINTS COMMAND

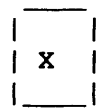
This command has the following format:



Type D to delete all breakpoints. Type the breakpoint number followed by D to delete breakpoint *n*.

15.3.15 SET COUNT REGISTER AND PROCEED FROM BREAKPOINT COMMAND

This command has the following format:



Type X to start execution at the current value of the P register; that is, to proceed with program execution. Execution continues until another breakpoint is encountered. (This command need not be followed by a carriage return.)

15.3.16 DISPLAY BUFFER MEMORY COMMAND

The debugger can also reference and modify Buffer Memory. The following command references a word of Buffer Memory.

Format:

```
 ____________
|            |
| Maddr/con1 |
|____________|
```

Type M followed by a Buffer Memory address (up to 8 digits), followed by a slash to display one 64-bit Buffer Memory word. A line feed displays the next sequential word.

A second form permits 16 words of Buffer Memory to be accessed at a time.

Format:

```
 ____________
|            |
| Maddr +|-  |
|____________|
```

Type M followed by a Buffer Memory address (up to 8 digits). If this is followed by a plus sign, the debugger displays the next sequential 16 words. Typing a minus displays the previous 16 words. Subsequently, + or - can be typed alone to scan memory in either direction. A carriage return or a line feed terminates the command.

Buffer Memory may also be accessed by the parcel.

Format:

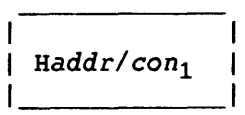
```
 ____________________________
|                            |
| Maddr A|B|C|D con1 [con2]  |
|____________________________|
```

Type M followed by a Buffer Memory address (up to 8 digits), followed by an A, B, C, or D to display 1 parcel of the specified address. The contents are modified by typing in the new value (con_2). A carriage return terminates the command. A line feed displays the next sequential parcel.

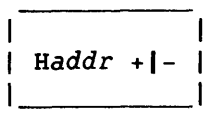
15.3.17 DISPLAY HIGH-SPEED CHANNEL COMMAND

The debugger can be used to reference and modify memory attached to any 100-Mbyte channel connected to the IOP. The console used must be attached to the IOP with the high-speed channel to be referenced. The commands are the same as those used by the Buffer Memory display commands, except that they are prefaced by an 'H' instead of an 'M'.

Formats:



Type H followed by the high-speed address to be displayed. The full 64-bit word is displayed in parcel format. A line-feed displays the next sequential word. A carriage return terminates the command.



Type H followed by a starting high-speed address. If followed by a +, 16 words beginning at the specified address are displayed. A - causes the previous 16 words to be displayed. Subsequently, + or - may be typed alone to scan memory in either direction. A carriage return terminates the command.



If the address is followed by an A, B, C, or D, the specified parcel is displayed. At this time, the parcel displayed can be modified by typing the new parcel in. A line feed displays the next sequential parcel. A carriage return terminates the command.

15.3.18 PROCESSING OF CHANNELS USED BY THE DEBUGGER COMMAND

The debugger preserves the state of the Done flag for the real-time clock, Buffer Memory, and console input channels. For a system being debugged, time effectively stands still while in the debugger, since the real-time clock interrupts are ignored.

15.4 PATCH OVERLAY

The overlay PATCH displays and/or modifies the value of a parcel in an overlay or in Kernel-resident code.

PATCH is activated by entering the following command at any Kernel console.

Format:

```
PATCH ovl|KERN addr [value]
```

ovl	Name of the overlay to be displayed or modified
KERN	Kernel code will be displayed or modified
addr	Parcel address, in octal, relative to the beginning of the overlay or of Kernel code
value	Optional octal value to enter into the specified address

If the optional value is not entered, PATCH displays the current contents of the specified location on the console screen, along with the name of the overlay (or KERN) and the address. If the optional value is entered, the new value is displayed.

In the following example, italic type represents the information displayed by PATCH:

```
PATCH ACOM 122
ACOM 000122 000027

PATCH ACOM 122 0
ACOM 000122 000000
```

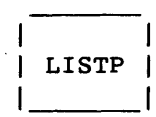
When an overlay is specified, PATCH searches the overlay area of Buffer Memory and displays or modifies the content of the address as directed. All overlays are released from Local Memory to ensure that the modified copy of an overlay is loaded from Buffer Memory the next time it is called.

If the Kernel is specified, PATCH searches the area of Local Memory containing the Kernel code.

15.5 LISTP OVERLAY

The LISTP overlay prints a listing of assigned $PUNTIF Kernel halt codes (in octal) and their definitions. LISTP is a Kernel overlay activated by entering the LISTP command at the MIOP console.

Format:



The following is an example of the output produced by LISTP.

KERNEL HALT (PUNT) CODES

Code	Meaning
000	No error code specified on $PUNTIF macro
001	Local Memory error (always hardware)
002	Buffer Memory error on deadstart (always hardware)
003	Buffer Memory error (always hardware)
004	High-speed channel error (always hardware)
005	Invalid message received from CPU
006	Invalid parameter in disk request from CPU
007	Program was executing at location 0
010	Local Memory location 0 was overwritten
011	Undefined message received on IOP communication channel
012	Overlay does not exist
013	Station stack overflow or underflow
014	Local Memory buffer not available
015	Buffer Memory disk buffer not available
016	Invalid local buffer release call
017	Buffer Memory incorrectly configured
020	IOP message channels incorrectly configured
021	SMOD is too large for area in Buffer Memory
022	Invalid Local Memory address
023	Illegal interrupt program sequence code
024	Stop request received from CPU
025	Low-speed channel error (always hardware)
026	Block number validation trap
040	Block multiplexer interrupt processor error
041	Bad CRW address in Device Table
042	Block multiplexer start I/O error
044	Block multiplexer configuration error
050	DD-49 disk software
051	Debugger was not loaded
052	Bad Buffer Memory allocation request
053	Bad Local Memory address on high-speed I/O call
054	Invalid I/O length specified

Code	Meaning
055	No high-speed channel configured for request
056	Illegal activity activation requested
057	Buffer Memory DAL queue exhausted
060	Illegal Demon call
061	Undefined Kernel service request
062	Illegal Kernel service request
063	Bad Kernel service request parameter
064	Requested device not configured
065	Illegal IOP requested on Kernel service request
066	Requested queue full
067	Illegal I/O address specified
070	SMOD error
071	Local Memory space exhausted
072	Illegal overlay load requested
073	Corrupted Local Memory chain
074	Bad Local Memory release
075	Bad I/O parameter
076	Unexpected interrupt received
077	Disk error
100	AMAP not available for system initialization
101	Illegal overlay number read during initialization
102	IOP initialization error
103	Buffer Memory configuration error
104	Overlay too large for loading
105	Premature tape end-of-file (EOF) encountered
106	Exit stack fault
111	Channel or buffer not configured
112	Path to memory not configured
113	Target memory not legal value

15.6 LISTO OVERLAY

LISTO is a Kernel overlay activated by entering the LISTO command at the MIOP console.

Format:



The overlay prints information on each overlay defined within the Kernel system. The information includes the following:

- Overlay number in octal
- Overlay name
- Size of the overlay specified in the octal number of parcels
- Octal address in Buffer Memory where the overlay is stored

Figure 15-3 shows an example of the output produced by LISTO.

No.	Ovl Name	Size	BM-Upr	BM-Lwr	No.	Ovl Name	Size	BM-Upr	BM-Lwr	No.	Ovl Name	Size	BM-Upr	BM-Lwr
00	AMAP	0454	00	017457	01	ACOM	2000	00	017572	02	AMSG	0744	00	020172
03	BMAGET	0374	00	020363	04	BMGET	0414	00	020462	05	CALL	1224	00	020565
06	CARD	0234	00	021032	07	CDEM	1100	00	021101	10	CONFIG	1254	00	021321
11	CRAY	0744	00	021574	12	DBGET	0360	00	021765	13	DISK	1600	00	022061
14	DISKIO	0470	00	022421	15	DKIOEX	0170	00	022537	16	DKSET	0274	00	022575
17	DKSET0	0220	00	022654	20	ERRECK	1704	00	022720	21	ERRDMP	1520	00	023301
22	FMGET	0410	00	023704	23	HDRPAG	1020	00	024006	24	HPLOAD	0454	00	024212
25	MASTER	0430	00	024325	26	MSTNEW	64	00	024433	27	HSPGET	0374	00	024450
30	OBIT	0700	00	024547	31	PATCH	0644	00	024727	32	START	0744	00	025100
33	START0	1610	00	025271	34	START1	1440	00	025633	35	START2	0760	00	026143
36	START3	0544	00	026337	37	START4	0334	00	026470	40	START5	0444	00	026537
41	STSGET	0404	00	026670	42	SYSS	3524	00	026771	43	SYSTEXT	0164	00	027716
44	TAPE	0510	00	027753	45	TCOM	0230	00	030075	46	TDUMP	1500	00	030143
47	TRACE	1330	00	030463	50	TRACK	0164	00	030751	51	USURP	0140	00	031006
52	XCRD	0364	00	031036	53	XPRT	0300	00	031133	54	SDMP0	2134	00	031213
55	SDMP1	1230	00	031642	56	SDMP2	1120	00	032110	57	SDMP4	0334	00	032334
60	SDMP5	0214	00	032423	61	SDMP6	0250	00	032466	62	SDMP7	0164	00	032540
63	SDMP8	0564	00	032575	64	BEGIN	34	00	032732	65	BLOCK	1530	00	032741
66	BTD	0260	00	033267	67	BTO	0224	00	033343	70	CHNTST	0764	00	033410
71	DEVICE	0400	00	033605	72	DIVIDE	0364	00	033705	73	DTB	0240	00	034002
74	LISTO	0660	00	034052	75	MSGHND	1014	00	034226	76	HPDATA	3770	00	034431
77	MULTIPLY	64	00	035427	0100	OTB	0150	00	035444	0101	PLOT	0510	00	035476
0102	PLOTIT	0724	00	035620	0103	PRNT	0204	00	036005	0104	TEST	0434	00	036046
0105	TIME	0510	00	036155	0106	UBTAPE	1524	00	036277	0107	UNBLK	0730	00	036624
0110	XCARD	0560	00	037012	0111	XCARDA	0730	00	037146	0112	XCARDB	0644	00	037334
0113	XPRINT	0760	00	037505	0114	XPRNTA	1400	00	037701	0115	XTAPE	1104	00	040201
0116	XTAPEA	1224	00	040422	0117	XTAPEB	1670	00	040667	0120	EDDELE	0120	00	041245
0121	EDINST	0550	00	041271	0122	EDPARS	0734	00	041423	0123	EDPRNT	0530	00	041612
0124	EDREPL	0120	00	041740	0125	EDTYPE	0234	00	041764	0126	LINGET	0344	00	042033
0127	LINPUT	0660	00	042124	0130	CLEAR	0230	00	042300	0131	COPY	1014	00	042346
0132	COPY0	1144	00	042551	0133	EDIT	1164	00	043002	0134	FDUMP	1170	00	043237
0135	FDUMP0	0304	00	043475	0136	FLOAD	0360	00	043556	0137	FLOAD0	1254	00	043652
0140	FSTAT	1004	00	044125	0141	PURGE	0464	00	044326	0142	DSKIO	0324	00	044443
0143	FILACC	0374	00	044530	0144	FILCLS	0164	00	044627	0145	FILCRE	0454	00	044664
0146	FILDEL	0414	00	044777	0147	FILGET	0330	00	045102	0150	FILNIT	0430	00	045170
0151	FILPOP	0274	00	023625	0152	FILPUT	0410	00	045276	0153	FILSTT	0234	00	045400
0154	RSTRT0	0744	00	045447	0155	RSTRT1	0454	00	045640	0156	RSTRT2	0564	00	045753
0157	SDMP3	0220	00	046110	0160	CONC	0760	00	046154	0161	CONCERR	0760	00	046350
0162	CONCO	1200	00	047112	0163	CONCI	1630	00	040544	0164	CRAYMSG	0440	00	047352
0165	ENDCONC	0574	00	047462	0166	ENTPID	0114	00	047621	0167	FREEBUFS	0340	00	047644
0170	LOGOFF	0240	00	047734	0171	LOGONA	0300	00	050004	0172	LOGONB	0544	00	050064
0173	LOGONC	0310	00	050215	0174	FEREAD	0240	00	050277	0175	FEWRIT	0124	00	050347
0176	CHKSM0	0170	00	050374	0177	CHKSM1	0240	00	050432	0200	MSGIO	0154	00	050502
0201	MSGIN	0730	00	050535	0202	MSGOUT	1204	00	050721	0203	REMVID	74	00	051162
0204	SPCHID	0104	00	051201	0205	TEXT	0764	00	051222	0206	UPDATE	0430	00	051417
0207	CONSL	1224	00	051525	0210	CLINIT	1304	00	051772	0211	LCP	0760	00	052253
0212	LOGON	1204	00	052447	0213	ONLINE	0460	00	052710	0214	POST	0434	00	053024
0215	PROTINIT	0774	00	053103	0216	SYNTAX	2030	00	053332	0217	QUEUE	0270	00	053740
0220	ACQTRM	54	00	054016	0221	ACQUIRE	0360	00	054031	0222	DISP01	1250	00	054125
0223	DISP02	0724	00	054377	0224	COMM01	1300	00	054564	0225	COMM02	1230	00	055024
0226	COMM03	0650	00	055272	0227	COMM04	0754	00	055444	0230	COMM05	1164	00	055637
0231	COMM06	0404	00	056074	0232	COMM07	0644	00	056175	0233	COMM08	0770	00	056346
0234	COMM09	1460	00	056544	0235	COMM10	1650	00	057060	0236	COMM11	0650	00	057432
0237	COMM12	0670	00	057604	0240	COMM13	0774	00	057762	0241	BABEL	1350	00	060161
0242	AMPEX	0550	00	060453	0243	HARDIS	1700	00	060605	0244	CLI	1350	00	061165
0245	COMBO	0130	00	061457	0246	CPUGET	0114	00	061505	0247	CRAYIO	0704	00	061630
0250	DECODE	1620	00	061711	0251	DECODB	0500	00	062255	0252	DESCRIBE	0164	00	062375
0253	DEVDAT	0540	00	062432	0254	DISPLAY	0614	00	062562	0255	DKDIS	0310	00	063135
0256	ERRDIS	1430	00	063007	0257	ERROR	0204	00	063315	0260	IFONSL	0270	00	063396
0261	IDEBUG	1024	00	063434	0262	IDLGET	0440	00	063641	0263	IDROT	1614	00	063751
0264	IFRMT	0524	00	064314	0265	JSTAT	1750	00	064441	0266	KEYBD	0504	00	065033
0267	LINK	1100	00	065154	0270	MESSAGE	0520	00	065374	0271	MSTAT	1514	00	065520
0272	MSTDIS	1044	00	066043	0273	OFFMT	0720	00	066254	0274	PROTOCOL	1214	00	066440
0275	READ	1450	00	066703	0276	STMSG	1614	00	067215	0277	SNAP	0520	00	067540
0300	SOPOC	0450	00	067704	0301	STADIS	1704	00	070016	0302	STAGEIN	1144	00	070377

1574

Figure 15-3. LISTO Sample Output

15.7 DKDMP OVERLAY

DKDMP is a Kernel overlay that dumps a selected portion of the disk directly to the IOS printer or Kernel console. The overlay is activated y entering the DKDMP command at the MIOP Kernel console.

Format:



DKDMP issues a sequence of messages requiring response. The following chronological list summarizes these messages and the required actions.

Message	Response/Action
-ENTER A TO REQUEST ABORT	Informative; typing an A in response to any of the following prompts terminates the disk dump operation.
-ALL ENTRIES ARE IN OCTAL	Informative; all responses to the following requests must be entered in octal.
PARCEL OR WORD FORMAT (P OR W)?	Enter "P" for parcel format, or "W" for word format; then press RETURN.
PRINTER OR DISPLAY (P OR D)?	Enter "P" to send the dump to the IOS printer, or "D" to send it to the Kernel console; then press RETURN.
IOP?	Enter the I/O Processor number to which the desired disk is attached and press RETURN.
CHANNEL?	Enter the desired channel number and press RETURN.
UNIT?	Enter the desired unit number and press RETURN. This message is only displayed when dumping data from a DD-39 or DD-40 disk unit. A response of 0 through 2 is legal for DD-39. A response of 0 or 1 is legal for DD-40.
CYLINDER?	Enter the desired cylinder number and press RETURN.
HEAD?	Enter the desired head number and press RETURN.

Message	Response/Action
STARTING SECTOR?	Enter the sector number at which to begin the dump and press RETURN.
NUMBER OF SECTORS?	Enter the total number of contiguous sectors to dump and press RETURN. The maximum number of sectors allowed is 177777_8. This message is only issued if the output is to the IOS printer.
CONTINUE (Y OR N)?	Enter "Y to view the next sector, or "N" to terminate. This message is only issued if the output is to the Kernel console.
-DISK DUMP COMPLETE	Informative; this message is displayed once all sectors are successfully dumped to the printer.
-CHANNEL *nn* EMPTY	Informative; this message is displayed when DKDMP determines that no disk is connected to channel *nn*. (This message is followed by the CHANNEL? request.)
-DISK DUMP ABORTED	Informative; this message is displayed when the program terminates early, either from an operator request (by entering an A) or from a disk or printer error.
-DISK ERROR	Informative; a disk read error has occurred. DKDMP aborts.
-INPUT PARAMETER ERROR	Informative; this message is displayed when the program detects an invalid operator response. This message is followed by another request for the parameter in which the error was detected. For example: INPUT PARAMETER ERROR CYLINDER?
-PRINTER ERROR	Informative; either the printer is not available or an error has occurred while the printer was trying to print a line. DKDMP aborts.

APPENDIX SECTION

A. DUMP ANALYSIS

This appendix describes some procedures used in analyzing I/O Subsystem (IOS) dumps:

- Record any operator interactions with the Kernel for later reference. Record the time and the date the dump was taken.

- Examine the punt code in the message. Get the meaning of coded dumps from LISTP (key-in) or from section 11, Debugging Tools.

- Take the dump according to CRI systems analyst specifications as follows:

  - Dump Kernel tables, DALS, non-Kernel tables and free memory, and Disk Control Blocks (DCBs).

  - It is advisable to take SYSDUMP and print the I/O Processor (IOP) that crashed. Other IOPs memories are then available if needed.

- Make sure the listing matches the dump. The assembly date in the listing should match date in Deadstart Done message. If they do not match, the approximate difference in the Kernel can be determined from the dump by finding the label INTERRUPT in the listing and comparing it to the first entry in the exit stack.

- Determine the overlay base address from operand register 3 (%B). If zero, the halt occurred within the Kernel-resident area.

- Determine the overlay number currently running from operand register 63 (%OVLNUM).

- Determine the overlay from the LISTO listing or by examining the address in operand register 3. (The first 8 characters in an overlay are its name in ASCII.)

- Get the exit stack (E) pointer from location EXSAVE+3. EXSAVE is a label in the Kernel immediately after the Kernel tables. Its format is as follows:

Bits	Description
0	Contents of A at halt
1	Carry (C)

Bits	Description
2	B register
3	Exit Stack pointer (E)
4-23	Exit Stack
24-163	DN + BZ (done and busy status) of all channels at halt

- Get P from EXSAVE+4+E-1. P is the address of a return jump or point in code when an interrupt occurred.

- If P is in an overlay, determine relative P by subtracting the base address (%B). (P is in an overlay if P is greater than %B.)

- Once the appropriate $PUNTIF macro is found in the listing, use the trace dump to determine the sequence of events leading up to halt (section 11, Debugging Tools, describes trace entries).

- Examine the following information in history traces:

  - The time of the trace entry in millisecond increments is saved, which gives a useful impression of the length of time between events.

  - The overlay in control at the time of the trace is always saved. If the Kernel is in control (usually in the idle loop), the overlay number is 177777.

  - On task activation traces ('TASK' - type 3), the address of the Activity Descriptor is saved. This information is useful to find which of the activities in the system is doing what and to analyze activity conflicts and possible timing problems.

  - Service requests to the Kernel are always traced and are usually helpful to understand the sequence of an activity's functions. In FDUMPs, they are printed as K-FNCT (in raw dumps they are type 5).

- If you suspect what is causing a halt (for example, A RELMEM), but you cannot find it in the trace (it may have happened too far in the past), try to re-create the halt, but with only one or two specific trace events enabled.

- Look for a pattern in the trace dump that would indicate that the system is looping through a group of overlays repeatedly. Try to isolate the reason for the loop.

- If the Kernel does not halt but seems to hang up and you are unable to access the system through the Kernel CRT, there may be an infinite loop in an overlay.

- Find the overlay in register 3 (%B).

- Examine the operand registers to determine the extent and reason for the loop.

- Although the done (DN) and busy (BZ) states of all channels are saved in the dump immediately after the exit stack and normally printed by FDUMP, you must know the following information when reading a raw dump:

  - If the channel is DN, the first parcel of the 2 parcels for this channel is set to 1.

  - If the channel is BZ, the second parcel is set to 1.

  - Accumulator channels from the IOPs for communication usually have the input channel busy (BZ).

  - The command channel for input from the Cray mainframe is normally awaiting input and is busy (BZ).

  - The CRT input channels are usually waiting for characters to be entered from the keyboard and thus are busy (BZ).

- Examine dedicated Kernel registers (described in table A-1) for irregularities.

Table A-1. Kernel Registers

Register	Value	Description
AA	0	Base address of the Kernel; always set to 0.
AB,AC	1,2	APML scratch registers
%B	3	Base address of the overlay; 0 if none.
%ACTIVE	6	Current Activity Descriptor, one in control; 177777 if none.
%SMOD	7	Address of SMOD currently being run; 177777 if none.
%OVLNUM	63	Pointer to current overlay number; 177777 if none.

Table A-1. Kernel Registers (continued)

Register	Value	Description
%PUNT	64	Address of Punt routine entrance
RD,RE	100,101	E at last interrupt; (E) at interrupt point where Kernel was last interrupted.
%CLKU,%CLKL	102,103	Kernel real-time clock (retained in milliseconds)
%ICHAN	106	Channel of last interrupt; zero if all interrupts processed.
%HISP	161	Channel number of high-speed (100- Mbyte) channel; zero if not present.

- Examine Activity Descriptors (AD) to see whether information in them is meaningful. The last AD activated has its address in %ACTIVE (operand register 6) if it is active. All ADs in the IOP are linked together through parcel 1 (AD@AL) of the ADs. The chain begins at EACT+1.

- Examine SMOD area in Local Memory to find registers at time of last service request to Kernel. Address of SMOD area is found in location ESMD in the Kernel tables. There is one SMOD area for each IOP, used only when an activity is active. If an activity is active, the address of the SMOD is located in %SMOD (operand register 7). Other SMODs are saved in Buffer Memory at an address found in the AD at location AD@MSU (parcels 3 and 4).

- Examine the following items in SMOD for good clues to problems:

  - AD address in Local Memory
  - Size of SMOD entry
  - Overlay table address of overlay making last call
  - A, B, C, and E registers
  - Exit stack
  - Operand registers saved on last CALL
  - Global registers if SMOD just read into memory

- For disk-related problems, examine Disk Control Blocks (DCBs) to see if any irregularities are evident. Note the following items:

  - Pointers to DCBs are found in Kernel tables DCCB through DCCB+17. (The last DCU-4 DCB to have any processing done is often in register DA. The last DCU-5 DCB to have any processing done is often in register DDCB defined in overlay DD49.)

  - DCU-4 channels that are being used have a nonzero value in parcel 0 (DB@FLG) and addresses of executable DALs are in parcel 2 (DB@EDL).

  - DCU-5 channels that are being used have a nonzero value in parcel 0 (DK@ACT).

  - The DCB contains a count of recovered and unrecovered errors on the channel and saved information about the last unrecovered error (such as cylinder, head, and sector).

B. IOS CONFIDENCE UTILITIES

I/O Subsystem (IOS) confidence utilities perform confidence tests on hardware elements of the IOS and report any errors. MOSTEST and HSPTEST take control of the IOS and cannot be run concurrently with other software. This appendix describes the commands in alphabetic order.

The following diagnostic commands can be entered at the Kernel console on any IOP:

Command	Description
CHNTEST	Tests 50-Mbit channels in loop-back mode
STOP	Halts execution of a test program

The following commands can be entered only at the MIOP Kernel console:

Command	Description
ECHOCP	Echoes data between the MIOP and the mainframe
HSPTEST	Tests 100-Mbyte channels
MOSTEST	Tests Buffer Memory
SSDTEST	Tests SSD Memory
XDK	Tests Peripheral Expander disk unit
XMT	Tests Peripheral Expander tape unit
XPR	Tests Peripheral Expander printer unit

The following command can be entered only at the XIOP Kernel console:

Command	Description
CPTEST	Tests on-line tape devices connected to the Block Multiplexer

B.1 CHNTEST COMMAND

The CHNTEST command initiates a basic channel loop back test to verify the reliable transfer of data on a 50-Mbit channel pair. The input channel must be cross-cabled to the output channel of the pair being tested.

CHNTEST is initiated by entering the CHNTEST command and channel number, followed by a carriage return, at the Kernel console of the IOP with the channels.

```
 _________________
|                 |
| CHNTEST chn     |
|_________________|
```

chn Channel number of the input channel of the pair

CHNTEST runs continuously until terminated by a channel error, data validation error or until the STOP CHNTEST command is entered at the Kernel console.

The Kernel console displays the following informational messages:

Message	Description
CHNTEST: INPUT ERROR - STATUS: xxx	An input channel error was detected. Channel status was xxx.
CHNTEST: INPUT ERROR - TIME-OUT	An input channel time-out error was detected.
CHNTEST: INPUT ERROR - NO READY/WAITING	An input channel error was detected. Ready/Waiting is not present.
CHNTEST: OUTPUT ERROR - STATUS: xxx	An output channel error was detected. Channel status was xxx.
CHNTEST: OUTPUT ERROR - TIME-OUT	An output channel time-out error was detected.
CHNTEST: DATA EXPECTED: yyy	A write data error was detected. Actual data was written to a 50-Mbit channel.

Message	Description
CHNTEST: DATA RECEIVED: zzz	A read data error was detected. Actual data was read from a 50-Mbit channel.
CHNTEST: TERMINATED	This is displayed when CHNTEST terminates

B.2 CPTEST COMMAND

CPTEST performs a minimum reliability test to on-line tape devices connected to the IOP Block Multiplexer. It writes a file to tape and reads it back using both read forward and read reverse. CPTEST performs no error correction and terminates on any error encountered.

CPTEST is initiated by entering the CPTEST command and parameters, followed by a carriage return, at the XIOP Kernel console.

```
CPTEST  U0 U1 . . . Un
```

U_n List of ordinals of devices to be tested

The devices should be configured with DOWN status. See the CONFIG command in the I/O Subsystem (IOS) Operator's Guide for COS, publication SG-0051 or the I/O Subsystem (IOS) Operator's Guide for UNICOS, publication SG-2005.

The XIOP Kernel console displays the following informational messages:

Message	Description
BAD DEVICE REQUEST	The device is already open or the device ordinal is invalid.
READ/WRITE ERROR	An I/O error occurred
CONTROL FUNCTION ERROR	An I/O error occurred on a Write Tape Mark, Rewind, or Backspace File command.
MOUNT FAILURE	The tape failed to become ready on the requested drive.

Message	Description
NO WRITE RING	The tape is file protected
READING	A tape is being read (forward direction)
WRITING	A tape is being written
REVERSE READING	A tape is being read (reverse direction)
CPTEST ABORTED	Displayed after another error message
OPEN FAILED	An unusual condition caused the tape mount to fail
CPTEST COMPLETE	Displayed if all functions were successful

B.3 ECHOCP COMMAND

The ECHOCP command initiates a test that echoes data between the MIOP and the Cray mainframe across the 50-Mbit channel, and validates data. The Cray operating system COS must be executing to run the test.

ECHOCP is initiated by entering the ECHOCP command, followed by a carriage return, at the MIOP Kernel console.

```
 _________
|         |
|  ECHOCP |
|_________|
```

ECHOCP runs continuously until terminated by a data validation error or until the STOP ECHOCP command is entered at the MIOP Kernel console.

The MIOP Kernel console displays the following informational messages:

Message	Description
ECHOCP: CPU NOT RESPONDING - RETRY?	The mainframe is not responding. Type Y to retry or N to terminate the test.

Message	Description
ECHOCP: DATA EXPECTED: yyy	A write data error was detected. Actual data written to the mainframe.
ECHOCP: DATA RECEIVED: zzz	A read data error was detected. Actual data read from the mainframe.
ECHOCP: CONTINUE?	Type Y to continue the test; N to terminate ECHOCP.
ECHOCP: TERMINATED	This is displayed when ECHOCP terminates

B.4 HSPTEST COMMAND

The HSPTEST command initiates a test that generates a high level of 100-Mbyte channel activity on each IOP with an attached 100-Mbyte channel to Central Memory. The option to validate data is available at the cost of reduced channel activity.

HSPTEST is initiated by entering the HSPTEST command, followed by a carriage return, at the MIOP Kernel console.

```
 ___________
|           |
|  HSPTEST  |
|___________|
```

Each MIOP Kernel console with an 100-Mbyte channel attached to Central Memory display the following prompting messages:

Message	Description
HSPTEST: (1) READ/WRITE/VERIFY	Read, write, and verify data.
HSPTEST: (2) READ/WRITE	Read and write data.
HSPTEST: (3) READ	Read data only.
HSPTEST: (4) WRITE	Write data only.
HSPTEST: (5) QUIT	Terminate HSPTEST.
HSPTEST: SELECT 1-5	Enter a number between 1 and 5 designating the type of channel activity.

Message	Description
HSPTEST: ENTER ADDRESS RANGE IN OCTAL	Enter a starting address followed by an ending address. The two addresses must be separated by at least one character that is not an octal digit. Addresses are checked to ensure that they are within the range of the configured memory size of Central Memory.
HSPTEST: (S)INGLE PASS OR (L)OOP	Type S for one pass, L for continuous execution.
HSPTEST: (B)LOCK OR (W)ORD INCREMENTAL TRANSFERS	Type B for one sector transfers (maximum channel rates), W for word incremental (address testing).
HSPTEST: STOP ON ERROR? (Y)ES OR (N)O	Type Y if test is to stop on error, N if test reports error and continues.

HSPTEST runs continuously until terminated by a STOP HSPTEST command entered at the IOP Kernel console or until an error is encountered during verification (if the stop on error option was selected). Progress of HSPTEST can be monitored on the station MONITOR HSPIO display.

The Kernel console displays following informational messages:

Message	Description
HSPTEST: DATA ADDRESS: xxx	A data error was detected. Central Memory address was xxx.
HSPTEST: DATA EXPECTED: yyy	A write data error was detected. Actual data was written to Central Memory.
HSPTEST: DATA RECEIVED: zzz	A read data error was detected. Actual data was read from Central Memory.

Message	Description
HSPTEST: CONTINUE?	This is displayed if the stop on error option was selected. Type Y to continue, N to terminate HSPTEST.
HSPTEST: NO HIGH-SPEED ON THIS IOP	This is displayed if the 100-Mbyte channel to Central Memory is not configured on this IOP.
HSPTEST: TERMINATED	Displayed when HSPTEST terminates

B.5 MOSTEST COMMAND

The MOSTEST command initiates a test that generates a high level of Buffer Memory I/O activity from configured IOP. The option to validate data is available at the cost of reduced channel activity.

MOSTEST is initiated by entering the MOSTEST command, followed by a carriage return, at the MIOP Kernel console.

```
 ___________
|           |
|  MOSTEST  |
|___________|
```

Each configured IOP Kernel console displays the following prompting messages:

Message	Description
MOSTEST: (1) READ/WRITE/VERIFY	Read, write, and verify data.
MOSTEST: (2) READ/WRITE	Read and write data.
MOSTEST: (3) READ	Read data only.
MOSTEST: (4) WRITE	Write data only.
MOSTEST: (5) QUIT	Terminate MOSTEST.
MOSTEST: SELECT 1-5	Enter a number between 1 and 5 designating the type of Buffer Memory I/O activity.

Message	Description
MOSTEST: (B)LOCK OR (W)ORD INCREMENTAL TRANSFERS	Type B for one sector transfers (maximum channel rates), W for word incremental (address testing).
MOSTEST: STOP ON ERROR? (Y)ES OR (N)O	Type Y if test is to stop on error, N if test reports error and continues.

MOSTEST runs continuously until terminated by a STOP MOSTEST command entered at the IOP Kernel console or until an error is encountered during verification (if the stop on error option was selected). Progress of MOSTEST can be monitored on the station MONITOR BMIO display.

The Kernel console displays the following informational messages:

Message	Description
MOSTEST: DATA ADDRESS: xxx	A data error was detected. Buffer Memory address was xxx.
MOSTEST: DATA EXPECTED: yyy	A write data error was detected. Actual data was written to Buffer Memory.
MOSTEST: DATA RECEIVED: zzz	A read data error was detected. Actual data was read from Buffer Memory.
MOSTEST: CONTINUE?	This is displayed if the stop on error option was selected. Type Y to continue, N to terminate MOSTEST.
MOSTEST: TERMINATED	This is displayed when MOSTEST terminates.

B.6 SSDTEST COMMAND

The SSDTEST command initiates a test that generates a high level of 100-Mbyte channel activity on each IOP with an attached 100-Mbyte channel to SSD Memory. The option to validate data is available at the cost of reduced channel activity.

SSDTEST is initiated by entering the SSDTEST command, followed by a carriage return, at the MIOP Kernel console.

```
 ____________
|            |
|  SSDTEST   |
|____________|
```

The Kernel console with an attached 100-Mbyte channel to SSD Memory displays the following prompting messages:

Message	Description
SSDTEST: (1) READ/WRITE/VERIFY	Read, write, and verify data.
SSDTEST: (2) READ/WRITE	Read and write data.
SSDTEST: (3) READ	Read data only.
SSDTEST: (4) WRITE	Write data only.
SSDTEST: (5) QUIT	Terminate SSDTEST.
SSDTEST: SELECT 1-5	Enter a number between 1 and 5 designating the type of channel activity.
SSDTEST: ENTER ADDRESS RANGE IN OCTAL	Enter a starting address followed by an ending address. The two addresses must be separated by at least 1 character that is not an octal digit. Addresses are checked to ensure that they are within the range of the configured SSD Memory size. The starting address must be on a 1000_8 word boundary. The ending address must be the last word of a 1000_8 word block.

Message	Description
SSDTEST: (S)INGLE PASS OR (L)OOP	Type S for one pass; L for continuous execution.
SSDTEST: STOP ON ERROR? (Y)ES OR (N)O	Type Y if the test is to stop on error; N if the test is to report errors and continue.

SSDTEST runs continuously until terminated by a STOP SSDTEST command entered at the IOP Kernel console, or until an error is encountered during validation (if the stop on error option was selected). Progress of SSDTEST can be monitored on the station MONITOR SSDIO display.

The Kernel console displays the following informational messages:

Message	Description
SSDTEST: DATA ADDRESS: xxx	A data error was detected. SSD Memory address was xxx.
SSDTEST: DATA EXPECTED: yyy	A write data error was detected. Actual data written to SSD Memory.
SSDTEST: DATA RECEIVED: zzz	A read data error was detected. Actual data read from SSD Memory.
SSDTEST: CONTINUE?	This is displayed if the stop on error option was selected. Type Y to continue, N to terminate SSDTEST.
SSDTEST: NO HIGH-SPEED ON THIS IOP	This is displayed if a 100-Mbyte channel to SSD Memory is not configured on this IOP.
SSDTEST: TERMINATED	This is displayed when SSDTEST terminates

B.7 STOP COMMAND

The STOP command terminates an on-line test. It must be entered in the following format at the Kernel console of each IOP in which the test is active:

```
 _____________
|             |
| STOP test   |
|_____________|
```

test On-line test name

B.8 XDK COMMAND

The XDK command initiates a confidence test on the Peripheral Expander disk unit. It writes, reads and compares data over the entire disk surface. Make sure that a scratch pack is mounted.

XDK is initiated by entering the XDK command, followed by a carriage return, at the MIOP Kernel console.

```
 _______
|       |
| XDK   |
|_______|
```

The test begins with the following query:

THIS TEST WRITES OVER THE ENTIRE DISK - CONTINUE?

A response of Y resumes the test.

XDK runs continuously until terminated by a STOP XDK command entered at the MIOP Kernel console or until an error is encountered during validation.

The Kernel console displays the following informational messages:

Message	Description
XDK: DATA EXPECTED: yyy	A write data error was detected. Actual data written to disk.
XDK: DATA RECEIVED: zzz	A read data error was detected. Actual data read from disk.

Message	Description
XDK: CONTINUE?	To continue type Y, N to terminate XDK.
XDK: TERMINATED	This is displayed when XDK terminates

B.9 XMT COMMAND

The XMT command performs a confidence test on the Peripheral Expander tape unit. It writes multiple files containing various data patterns, then reads back and compares data.

XMT is initiated by entering the XMT command, followed by a carriage return, at the MIOP Kernel console.



XMT runs continuously until terminated by a STOP XMT command entered at the MIOP Kernel console or until an error is encountered during validation.

The Kernel console displays the following informational messages:

Message	Description
XMT: DATA EXPECTED: yyy	A write data error was detected. Actual data was written to tape.
XMT: DATA RECEIVED: zzz	A read data error was detected. Actual data was read from tape.
XMT: CONTINUE?	To continue type Y, N to terminate XMT.
XMT: TERMINATED	This is displayed when XMT terminates.

B.10 XPR COMMAND

The XPR command performs a confidence test on the Peripheral Expander printer unit. It writes alternate pages in alpha and plot mode, using various data patterns, and compares data.

XPR is initiated by entering the XPR command, followed by a carriage return, at the MIOP Kernel console.



XPR runs continuously until terminated by a STOP XPR command entered at the MIOP Kernel console or until an error is encountered.

The Kernel console displays the following informational message:

Message	Description
XPR: TERMINATED	Displayed when XPR terminates

C. SYSDUMP

SYSDUMP is the activity that is invoked to dump selected areas of the system to disk for examination.

SYSDUMP operates with few dependencies on the I/O Subsystem (IOS). However, SYSDUMP operation requires that the following be intact:

- The routine in Master I/O Processor (MIOP) that initializes SYSDUMP
- SYSDUMP routines (must be in Buffer Memory)
- Overlay tables
- The pointer to the Overlay Table

C.1 OPERATIONAL DESCRIPTION

In this subsection, SYSDUMP routines (SDMP0-SDMP10) are discussed in operational sequence as follows.

Routine	Operational Sequence
SDMP0	1. CONTROL-D is sensed by the keyboard interrupt handler; otherwise, a PUNT (system-controlled crash) occurs.
	2. A, C, B, E, and the exit stack, as well as the Channel Done and Busy flags, are saved.
	3. A routine is called, within the operating system, which loads SDMP0 overlay into MIOP Local Memory above the register save area.
	4. SDMP0 displays the default parameter list and queries for changes. Once the parameters have been established, they are written out to Buffer Memory. SDMP0 then loads SDMP1 into MIOP Local Memory at the same starting address of SDMP0 and passes it control. (SDMP1 is shorter in length than SDMP0 so that the overlay load routine at the end of SDMP0 can complete successfully.)

Routine	Operational Sequence
SDMP1	1. SDMP1 sends a message to the I/O Processor (IOP) that has the master disk attached to it, informing that IOP of the SYSDUMP. (SYSDUMP currently supports the master disk attached to BIOP only.) 2. The IOP receiving the message saves its registers as did MIOP. 3. The IOP loads in SDMP2 above the register save area and passes control to SDMP2. Figure C-1 shows the memory map during SYSDUMP of the IOP that has the master disk attached to it. (IOP-1 or -2, depending on number of IOPs attached).

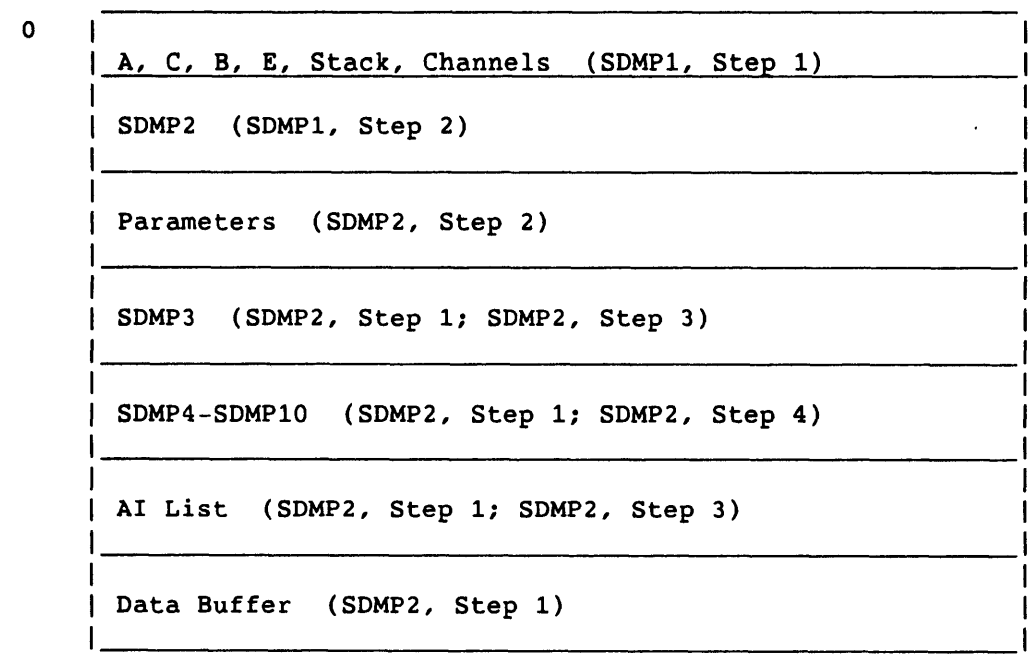


Figure C-1. SYSDUMP Memory Map of IOP with Master Disk Attached

Routine	Operational Sequence
SDMP2 (BIOP)	1. SDMP2 sets up four areas in its IOP, one for each of the following: • The disk driver; SDMP3 for DD-19 or DD-29 disks, SDMP3A for DD-39 and DD-49 disks. • SDMP4-SDMP10, which are loaded one at a time to dump various areas of the system • A buffer used to contain the AI list (see figure C-2) read in from the master device. This list describes the sectors on the master device that have been reserved for the dump. • A data buffer, which is loaded by the various dump routines to be written out to disk 2. SDMP2 reads in the parameter list from Buffer Memory. 3. SDMP2 loads in the disk driver and searches for the label on the master device. Once found, it uses the AI in the label to find the start of the SYSDUMP area. The first sector of this area contains a list of all AIs reserved for SYSDUMP. Each AI points to a physical track on the disk. The AI is decoded by dividing the AI by the number of heads in a cylinder. The quotient is the physical cylinder number and the remainder is the physical head number. The first AI in the list always points to the track that contains the AI list itself. The SYSDUMP always begins on sector 1 of the first AI. The AI list is terminated by a binary 0.

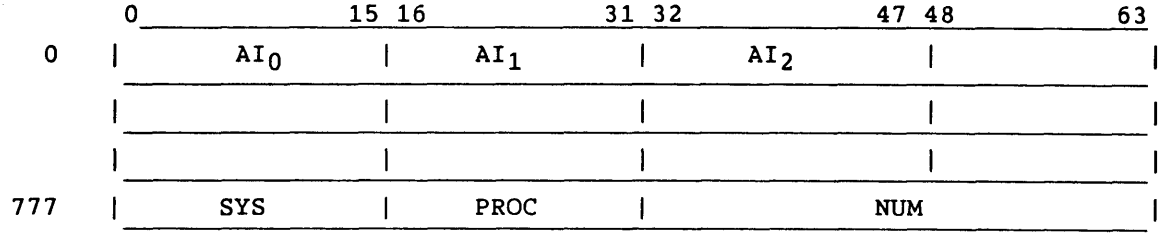


Figure C-2. AI List

Routine	Operational Sequence

The last word of the AI list is used to describe the dump to COS startup. This information is transcribed into a dump header attached to the new version of the permanent dataset 'CRAY1SYSTEMDUMP' (see FDUMP in the Operational Aids Reference Manual, publication SM-0044).

The header area for SYSDUMP, when running under UNICOS, contains an ASCII ID followed by a list of partitions relative to the starting cylinder. Each partition consists of a starting block number and a count of blocks in that partition. A zero in the partition count field terminates the list. The last word of the header is formatted as it is for COS (figure C-3).

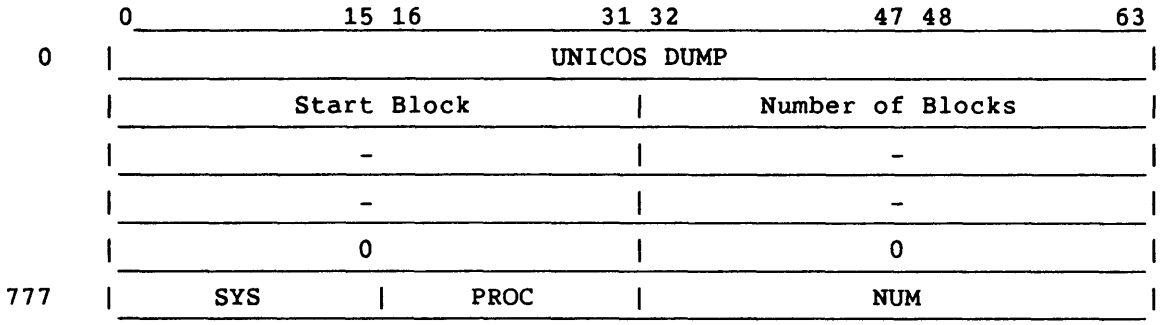
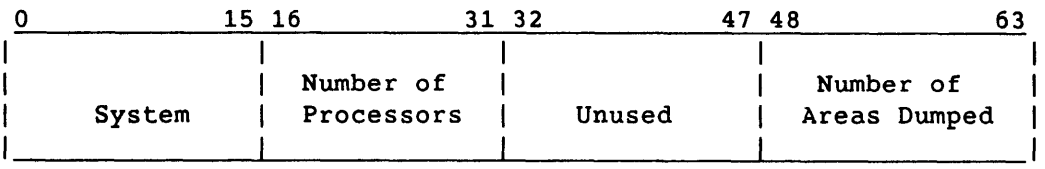


Figure C-3. Head Format for OS = UNICOS

The format of the dump header is as follows:



Bits	Description
0-15	System type: 0 SYS$S CRAY-1 S or CRAY-1 M 1 SYS$X CRAY X-MP 3 SYS$YX CRAY X-MP EA 4 SYS$Y CRAY Y-MP

Routine	Operational Sequence	
	Bits	Description
	16-31	Number of processors in system
	32-47	Unused
	48-63	Number of areas dumped (number of control words)

4. SDMP2 loads in each of the dump routines SDMP4-SDMP10 to dump its portion of the system.

SDMP4 (Central Memory and Registers)

SDMP4 dumps Central Memory and the CPU registers as follows:

1. A message is sent to SDMP1 in the MIOP requesting that the Exchange Packages in each processor be saved in a fixed area of memory so that they can later be found by dump formatting routines. SDMP1 then loads a routine (SDMPA for the CRAY X-MP, CRAY-1 S, or CRAY-1 M computer systems; SDMPYA for the CRAY X-MP EA or CRAY Y-MP computer systems) into Central Memory to save the Exchange Packages.

2. SDMP4 moves the specified areas of memory across the high-speed (100-Mbyte) channel to Buffer I/O Processor (BIOP) Local Memory, from where it is written to disk.

3. SDMP4 sends a request to SDMP1 that CPU registers be moved to fixed locations in Central Memory. SDMP1 then loads a routine (SDMPB for the CRAY X-MP, CRAY-1 S, or CRAY-1 M computer systems; SDMPYB for the CRAY X-MP EA or CRAY Y-MP computer systems) into Central Memory to save the CPU registers.

4. SDMP4 moves the registers across the high-speed channel and writes them to disk.

5. SDMP4 moves specified areas of system and user memory across the high-speed channel to BIOP Local Memory from where it is written to disk.

Routine	Operational Sequence
	Dumping of these areas is controlled by the TABLE MEMORY dump parameter. If the parameter is YES, location 201_8 is Central Memory is expected to contain a pointer to a table of memory addresses and lengths of areas to be dumped. The pointer is one word. The high-order 32 bits are the address of the table; the low-order 32 bits are the number of entries in the table. Each table entry is one word. The high-order 32 bits are the address of an area to be dumped; the low-order 32 bits are the number of words to dump. If the length of any entry is 0, the corresponding area is skipped. Each area is dumped as Central Memory type (MM$CRAY).
SDMP5 (Buffer Memory)	SDMP5 dumps the selected areas of Buffer Memory to disk. The selected areas are read from Buffer Memory to Local Memory, from where they are written to disk.
SDMP6 - IOP-0 (MIOP)	SDMP6 dumps IOP-0 (MIOP) along with its registers to disk as follows: 1. SDMP6 sends a request to SDMP1. As a result, SDMP1 writes out the entire MIOP followed by its registers to Buffer Memory. 2. SDMP6 moves this data from MOS to its Local Memory, from where it is written to disk.
SDMP7 - IOP-1 (BIOP)	SDMP7 dumps IOP-1 (BIOP) and its registers to disk as follows: 1. SDMP7 writes Local Memory directly to disk. 2. It then records the registers in memory and writes them to disk.

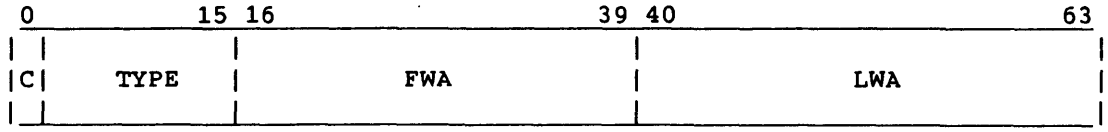
Routine	Operational Sequence
SDMP8-IOP-2 /IOP-3(DIOP) /(XIOP)	SDMP8 dumps IOP-2 (DIOP) and IOP-3 (XIOP) to disk as follows: 1. SDMP8 dead dumps the IOPs to Buffer Memory, where they can then be read into BIOP Local Memory and be written to disk. 2. SDMP8 loads a routine into each of the two IOPs, causing them to record their registers to Local Memory and then write them out to Buffer Memory. 3. SDMP8 then reads these registers in from Buffer Memory and writes them to disk.
SDMP9 - SSD	SDMP9 dumps the selected areas of the SSD to disk as follows: 1. A request is sent to SDMP1 in the MIOP to have the CPU move a 40K block of the SSD to Central Memory. SDMP9 writes the starting address of the 40K block to Buffer Memory. 2. SDMP1 reads in the address and passes it to a routine (SDMPD for the CRAY-1 S or CRAY-1 M computer system, SDMPE for the CRAY X-MP computer system, or SDMPYD for the CRAY X-MP EA or CRAY Y-MP computer systems) that is loaded into Central Memory so that it can execute in the CPU. 3. SDMP9 then moves the SSD data across the high-speed channel to Local Memory, from where it is written to disk.
SDMP10 - Cluster Sets	SDMP10 dumps the cluster register sets to disk as described below. Discussion of cluster registers applies only to the CRAY X-MP or CRAY Y-MP computer systems. The number of cluster register sets is the number of processors plus 1. 1. SDMP10 sends a request to SDMP1 in the MIOP to move the cluster register sets into Central Memory. 2. SDMP10 loads a routine (SDMPC for the CRAY-1 S, CRAY-1 M, or CRAY X-MP computer systems; SDMPYC for the CRAY X-MP EA or CRAY Y-MP computer systems) into Central Memory so that it can execute in the CPU.

Routine	Operational Sequence
	3. SDMP10 reads the cluster registers in across the high-speed channel and writes them to disk.

C.2 DUMP FORMAT

Each area dumped begins on a sector boundary. The first word of each area dumped is a dump control word (DCW).

Format:



Field	Word	Bits	Description
C	0	0	Compressed code; always 0.
TYPE	0	1-15	Type of area dumped is as follows: 0 Central Memory (MM$CRAY) 1 Buffer Memory (MM$MOS) 2 Cray registers-CPU0 (MM$CREG) 3 IOP-0 (MM$A0) 4 IOP-1 (MM$A1) 5 IOP-2 (MM$A2) 6 IOP-3 (MM$A3) 30-37 SSD (MM$SSD) 41-57 Cray registers-CPU1 (MM$CREG1) 100-137 Cluster registers (MM$CLUS)
FWA	0	16-39	First word address; dependent on type (see LWA).

Field	Word	Bits	Description
LWA	0	40-63	Last word address; dependent on type, is as follows:

TYPE	FWA	LWA
0,1	First word	Last word
2-6, 41-57, 100-137	Number of words	0
30-37	First 512-word block	Last 512-word block

Areas of system lost due to SYSDUMP processing areas follows:

Buffer Memory:	0 through 10,000
CPU:	0 through 100 plus 20 words per processor

D. ISP CHANNEL DRIVER

This appendix describes the structure of the Integrated Support Processor (ISP) channel driver (ISPDRV). ISPDRV is a driver overlay, callable by the User Channel shell in response to F-packet requests from the CPU. ISPDRV drives Cray Front-end Interface (FEI) devices on Master I/O Processor (MIOP) low-speed channels for I/O to the ISP. See section 9, User-channel I/O, for more information on the User Channel Shell. The ISP is an IBM or IBM-compatible mainframe with peripheral devices that can be accessed directly by COS jobs. ISPDRV requests originate in STP task IQM.

The ISP channel driver is a single, self-contained overlay which relies on the UCSHL overlay for all its functions except actual I/O to the FEI device.

D.1 MAIN LOOP

The ISPDRV main loop executes until operand register R!END is nonzero. This is set during the processing of a Driver Close request, so the life of the main loop is from open to close. The overlay ends with a TERM request at the end of the main loop.

The loop begins with a WATCH macro that gets the next command code in register CMD from the shell. At the bottom of the loop, it performs a SIGNAL to send the driver status code (R!UST) back to the shell.

Between the WATCH and the SIGNAL, the main loop examines the command code (R!CMD) and calls a subroutine to process it.

D.2 OPEN PROCESSING

Driver Open commands are handled by subroutine DOOPN, which is called by the main loop. The Driver Open command is always the first one given to the overlay by the shell. DOOPN initializes operand registers, master-clears the FEI device, and saves the Double-buffer flag from the request packet from the CPU.

D.3 CLOSE PROCESSING

Driver close is always the last command sent to the overlay by the shell because ISPDRV terminates if and only if it receives a Driver Close command. Close processing consists only of setting the status complete (R!UST = UC$CMPT) and setting R!END nonzero to force termination by the main loop. This is done by subroutine DOCLS.

D.4 I/O REQUEST PROCESSING

All I/O commands (UC$RD, UC$RDL, UC$WRT, and UC$WRTL) are handled by subroutine DOIO, called by the main loop. DOIO consists of two subloops: one for single-buffered I/O and one for double-buffered I/O. The double-buffer loop is at label XIO; the single-buffered loop is at label SIO. DOIO enters one or the other based on the value of the Double-buffer flag saved from the Driver Open request. The two loops contain the same basic steps, but in a different order, as table D-1 shows.

Table D-1. Stepflow for DOIO Buffered Loops

XIO		SIO	
R = STARTIO	.START I/O	R = STARTIO	.START I/O
SIGNAL	.RETURN STATUS OF LAST REQ	R = WAITIO	.WAIT DONE
WATCH	.PREFETCH NEXT REQUEST	SIGNAL	.RETURN STATUS OF THIS REQ
R = WAITIO	.WAIT DONE	WATCH	.GET NEXT REQ

XIO signals a dummy status code (UC$NOOP) on the first pass to get the next request from the shell without waiting for the first block of I/O to complete. On subsequent passes, XIO fetches shell requests without waiting for I/O completion. It does not have to wait for the Shell before starting the next I/O. This can be done in double-buffered mode because the shell can transfer one buffer's contents to or from the CPU while the driver is using the other CPU for I/O.

SIO handles I/O commands synchronously, starting and waiting for completion of each I/O before signaling the shell for another.

Both loops terminate when either STARTIO or WAITIO puts an ending status code in R!UST. The loops exit before the SIGNAL request, and the main loop signals the final status to the shell.

D.5 STARTIO SUBROUTINE

STARTIO is responsible for beginning I/O operation on the FEI device, and holding the disconnect on all but the last block on output.

STARTIO also does some special processing for output channels if the last write resulted in a time-out. In this case, it checks the channel status register and waits for the Sequence Error flag (indicating that ISP has a Read request ready on this channel) before starting I/O. If the Sequence Error flag is not raised during the time-out interval, STARTIO generates time-out status (R!UST = IE$TMO). STARTIO waits with PAUSE requests to the Kernel.

The purpose of this wait for Sequence Error is to clean up the FEI device after an output time-out without generating unsolicited interrupt signals to the IBM channel. After an output time-out, buffers in the FEI contain a fragment of data which is read by the ISP the next time it reads the channel. The ISP must discard this fragment and reissue the Read request in order to recover from the time-out. Before restarting output, STARTIO sends a master clear signal to the FEI after every time-out. This causes the FEI to give unit check status to the IBM channel. The wait for Sequence Error makes the unit check appear while the ISP is reading the channel, and the interrupt is not unsolicited. Unsolicited interrupts are processed by MVS, which does not know what to do with this foreign device, and it hangs the IBM CPU until the operator clears the channel.

D.6 WAITIO SUBROUTINE

WAITIO performs two functions: waiting for the current I/O to complete, and processing the completion.

WAITIO waits with a TPUSH request to the Kernel, if an interrupt from the channel is not already pending; the UC@IRT flag in the UCT tells it if it is pending. WAITIO regains control if the Kernel receives an interrupt on the channel or if the timer interval expires (time-out).

When the I/O is complete, WAITIO checks it for errors and determines the length of the data transfer.

If the I/O times out (UC@IRT = 0 after the TPUSH), WAITIO checks to see whether data has begun moving by reading the current channel address. If it has, it pauses 100 ms more to allow the transfer to complete. This pause avoids false time-outs which would interrupt a transmission in the middle. On a time-out, WAITIO sets error status (R!UST = IE$TMO) which forces the I/O loop to terminate.

D.7 GETL SUBROUTINE

Subroutine GETL contains the only code in ISPDRV that is dependent on the ISP channel protocol. It is necessary because the protocol allows for variable-length blocks of data on the channel. The CPU issues read requests for the maximum amount of data it expects to receive; the actual length is generally less, and it is the job of ISPDRV to detect the end of the block and tell the CPU the actual length.

Due to the low-speed channel architecture, it is impossible for ISPDRV to detect the end of a block that is an exact multiple of its buffer size (512 Cray words, a very common size in ISP transmissions). If an incoming block ends at the same time the buffer is filled, ISPDRV would continue reading, receiving the next block as part of the current one, if it did not otherwise know how much data to expect. For this reason, subroutine GETL is included in the driver.

GETL examines input blocks when channel headers are expected, and saves the block length fields from the header. Based on these fields, WAITIO can return completion status to the shell when the block actually ends. A certain amount of validation in GETL is necessary to ensure that it is actually looking at a channel header; this validation is limited to verifying that the 16-bit checksum of the first 48 bytes is 0 (a convention of the ISP protocol) and that the channel header length field contains the correct value of 48 bytes.

INDEX

INDEX

$APTEXT constants, 3-9
$B@CB, 6-7
$ELSE macro, 14-12
$ELSEIF macro, 14-12
$ENDTIL macro, 14-13
$GOTO macro, 14-14
$IF macro, 14-11
$PUNTIF macro, 14-15
$R@CLI, 6-7
$T@KEY, 6-7
$UNTIL macro, 14-13
B operand register, 15-6
%CLI register, 6-3
%EX operand register, 2-13
%LIMIT global registers, 6-4
%MSEC (interval counter), 2-49
%PROT, 6-3
%STACK global register, 6-4
%STAT register, 6-3
%STCON register, 6-3
%TENTHS (1-second interval counter), 2-50
0000*nn* trace event parameter, 15-10 to 15-12, 15-19, 15-22
1-second interval counter (%TENTHS), 2-50
177*nnn* accumulator message, 2-54
100 Mbyte channel
    channel test, B-5, B-9
    definition, 1-7
    specified, 1-3
50-Mbit channel pair loop back test, B-2, B-4
6 Mbyte channel
    specified, 1-3
    definition, 1-7

A register (accumulator), 1-2
A-Asnd trace event parameter, 15-9, 15-15 to 15-16
A-to-A message passing, 4-5
A-TO-A trace event parameter, 15-9, 15-15, 15-16
A130 adapter, 10-1
A130OI function, 2-21
ABORT interactive command, 8-8
Abort Transfer Request (ATR), 3-23
Accumulator (A register), 1-2
    177*nnn* message, 2-54
ACOM
    overlay, 3-2 through 3-8
    trace event parameter, 15-16
Acomsg trace event parameter, 15-9, 5-16
Activate
    activity in another IOP, 15-13
    overlay (GOTO), 2-29
Active overlays displayed by SUMMARY utility, 15-3
Active stream count (AST), 6-2
Activity
    Descriptor (AD), 2-7
    dispatching
        exit from, 15-8
        parameters, 2-1
    software stacking, 2-7
    termination, 1-7
    vs task, 1-7
AD (Activity Descriptor), 2-7
AD@P1, use with CREATE function, 2-24
Adding an overlay, 14-2
ADDRESS macro, 14-19
ADEM, 10-1
    common packet handling demon, 9-5
    FEI logical path overlay, 11-1
    NSC overlay, 10-6
    VMEbus overlay, 13-2
Advance
    command (KIC$AC), 5-24
    data (KIC$AD), 5-25
AI list, C-3
ALERT
    function, 2-16
    stepflow, 2-16
    trace event parameter, 15-13
Allocate
    DAL, 2-27
    Local Memory, 2-1, 2-27, 15-14
Allocation/deallocation, memory, 2-12
Allow overlay to send message packet, 2-45
AMAP overlay
    channel declaration, 9-10
    memory parameters in, 2-1, 2-10
$APTEXT constants, 3-9
AR@, 3-23
ASLEEP function, 2-18
Assign
    control-unit, 5-18
    device path by BMXAIO, 5-21
    device path by BMXSIO, 5-17
Assignments, register, 1-6
ATR (Abort Transfer Request), 3-23
ATTENTION interactive command, 8-8
AWAKE function
    description, 2-19
    stepflow, 2-19

AWAKE function (continued)
trace event parameters, 15-13
used with ALERT, 2-16

$B@CB, 6-7
B-packet, 10-5
Backspace record (FC$BKSPC), 4-42
Backward space file (FC$BKFIL), 4-42
Bank, 5-3
BCOM trace event parameter, 15-17
BCOM0
description, 4-5
BCOM1
description, 4-5
in read requests, 4-13
in write requests, 4-27
BCOM3
description, 4-5
in end read requests, 4-37
in free requets, 4-59
in load display requests, 4-45
in mount requests, 4-6
in no-op requests, 4-41
in positioning requests, 4-43
in read requests, 4-10, 4-19
in remount requests, 4-48
in request and response routing, 4-5
in rewind requests, 4-51
in unload requests, 4-54
in write requests, 4-23, 4-33
Bcm-rs trace event parameter, 15-10, 15-17
Bcomsg trace event parameter, 15-10, 15-17
BDV@ Device Table, 5-7, 5-9
Bfm-dn trace event parameter, 15-10, 15-18
Bfm-in trace event parameter, 15-10, 15-17
BGET function, 2-22
BIOP responsibilities, 1-5
(or DIOP) data handler (TDEM1), 4-19
Block Multiplexer (BMX) subsystem, 4-1, section 5 (See also BMX)
interface software, 1-2
interface, 5-1
interface routines, 5-12
Blower air operator message, 3-38
BMX (IOS Block Mux subsystem), section 5
channel driver, 5-1, 5-21
Channel Look-up Table (XCHT), 5-26
channel program word, 5-9
buffer memory transfer, 5-10
command chaining, 5-11
local memory, 5-10
nondata transfer, 5-9
configuration, 5-3
interrupt handler (IBMX), 5-26
overview, 5-2
routines, 5-12
BMXCON, 5-12
BMXCPU, 5-15
BMXSIO, 5-16
BMXAIO, 5-20
BMXDEM, 5-21
routines (continued)
IBMX,, 5-26
BMXOPE, 5-28
tables, 5-3, 5-8
BMX*n* trace event parameters, 15-9 to 15-10, 15-16 to 15-17
Bmx-*nn* trace event parameters, 15-9 to 15-10, 15-16 to 15-17

BMXAIO, 5-20
assign device path, 5-21
halt I/O, 5-21
release device path, 5-21
request reset, 5-21
BMXCON
channel configuration, 5-13
control unit configuration, 5-13
description, 5-13
device configuration, 5-14
messages, 5-14 to 5-15
overlay, 5-1
states, 5-12
BMXCPU overlay, 5-1
BMXDEM
advance command sequence, 5-24
advance data sequence, 5-25
description, 5-21
overlay, 5-1
request-in sequence, 5-25
start command sequence, 5-22
trace event parameter, 11-16, 11-17
BMXOPE overlay, 5-1
in free requests, 4-59
in mount requests, 4-7
in remount requests, 4-48
open, 5-28
close, 5-28
BMXSIO
overlay, 5-1
return to caller, 5-20
start I/O, 5-16
trace event parameter, 15-17
wait I/O, 5-20
BMXTPO
description, 5-2, 5-29
in mount requests, 4-8
in remount requests, 4-49
Breakpoints, 15-22
BRET function, 2-23
Buffer I/O Processor, 6-1
Buffer Memory
addresses requirements , 2-3
Control Block (MCB), 3-23
data transfer commands, 5-10
I/O activity test, B-7
management, 3-24
meaning, 1-7
organization, 2-4
resident datasets, 2-6
return to pool, 2-36
usage, 2-3
Buffering, shell, 9-9

Buffers
    disk, 3-1
    I/O, 2-1
BUFMAN routine
    description, 4-2, 4-3
    in end read requests, 4-39
    in free requests, 4-59
    in load display requests, 4-46
    in no-op requests, 4-41
    in positioning requests, 4-43
    in read requests, 4-11 to 4-12, 4-23
    in rewind requests, 4-52
    in unload requests, 4-55
    in write requests, 4-25 to 4-26
    trace event parameter, 15-17, 15-18
BUGOFF command, 15-22
Bus-out check, 4-62
BYE interactive command, 8-8
Bypass
    channel, issue I/O on, 15-19
    mode, 1-3
BYPASS activity
    description, 4-2
    in end read requests, 4-37 to 4-38
    in free requests, 4-59
    in no-op requests, 4-41
    in read requests, 4-10 to 4-11, 4-12 to 4-13
    in write requests, 4-23 to 4-25, 4-26 to 4-27
BYPIO trace event parameter, 15-10, 15-19

C$LOCK, 6-7
CALL
    function, 2-23
    trace event parameter, 15-14
Call overlay (OUTCALL), 2-38
    to perform a function, 15-14
Carriage return, 1-8
CB@, 3-23
CBT@ (Control-unit Bank Table), 5-8, 5-15
CDEM overlay, 3-3, 4-5
Central Memory
    meaning, 1-7
Chain, free memory, 2-12
Chaining
    command, 5-11 to 5-12
    sector, 3-24, 3-26
Chains, Local Memory, 2-12
CHANGE interactive command, 8-8
Chan-io trace event parameter, 15-9, 15-12 to 15-13
Channel
    configuration (CON$CHN), 5-13
    control unit message format, 5-14
    driver, BMX subsystem, 5-21
    Extension Table, 10-12
    ID ordinal description (NSC activity), 10-12
    interrupts displayed, 15-3
    message format, 5-14
Channel (continued)
    mode, 5-23
    Program Word (CPW), 5-9
    queue (XCIQ), 5-22
    table list, 5-8
    Tables (CHT@), 5-8
    tables for each configured channel, pointer to, 5-7
    time-out, 5-24
CHT@ Channel Table, 5-8
CLEAR macro, 14-31
CLI task, 6-11
    interaction, 6-13
CLOCK demon, 2-49
Clock-event time-out, 15-19
Close
    HSX request, 12-3
    processing, ISP, D-2
    request (CR$CLS), 9-3
    subroutine, UCSHL, 9-6
CNT (Configuration Table), 5-1, 5-3
Command
    buffer memory data transfer, 5-10
    chain mode, 5-23
    chaining, 5-11 to 5-12
    local memory transfer, 5-10
    nonlocal data transfer, 5-9
    sequences for HSX, 12-9 to 12-10
    skip data transfer, 5-24
    parameters global symbol, 6-33
Commands
    ABORT, 8-8
    ATTENTION, 8-8
    BUGOFF, 15-22
    BYE, 8-8
    CHANGE, 8-8
    COMMENT, 8-8
    END, 8-3
    EOF, 8-8
    ERRDMP, 2-59
    LOG, 8-1
    LOGOFF, 8-3
    LOGOFF, 8-8
    LOGON, 8-8
    POLL, 8-3
    STATUS, 8-8
    TEST-I/O, 5-13
    debugger, 15-22
    interactive, section 8
    on-line TRACE, 15-4
COMMENT interactive command, 8-8
Common
    deck structure, 14-2
    decks, 14-1
    interrupt handler, exit from, 15-8
    packet handling demon (ADEM), 9-5
Communication
    among IOPs, 2-52
    channel, MIOP-mainframe, 2-53
    errors, 6-1
    initialization, MIOP-mainframe, 2-55
    packets (DALs), 2-1

Computation section, IOP, 1-3
Computer
    maintenance, 2-59
    front-end, 1-2
CON$, 5-12
CONC overlay description, 7-2
Concentrator
    errors, 7-7
    initialization, 7-2
    interactive, 8-1
    overlays, interactive, 8-1
    software, 1-2
       tree structure, 7-1
    status, 15-11, 15-22
    termination, 7-8
CONCERR overlay description, 7-7
CONCID overlay description, 7-7
CONCIO activity description, 7-2
Concurrent disk I/O requests, 3-1
Confidence utilities, appendix B
Configuration
    block mux, 5-2
    change request (FC$CHANGE), 4-6
    device, 5-14
    I/O Subsystem, 1-2
    maps, 2-1
    states (CON$), 5-12
    Table (CNT), 5-1, 5-3
    User Channel, 9-10
CONMAN activity in configuration change
  requests, 4-6
CONSOLE command, 6-1
Console
    interactive, 8-1
    output, 6-34
    overlays, interactive, 8-6
Contents, System Directory, 2-3
Contingent connection, 5-18
Continue Request-in (KIC$CR), 5-28
Control
    logic, 1-2
    request for HSX channel, 12-2
    section (IOP), 1-3
    unit configuration (CON$CUT), 5-13
    Unit Tables (CUT@), 5-8
    words, 4-4
Control-unit
    assign, 5-18
    Bank Tables (CBT@), 5-8, 5-17
    busy, 5-19
    path already assigned, 5-18
Controlling disk software
    DD-19 and DD-29, 3-2
    RD-10, DD-39, DD-40 and DD-49, 3-21
Conventions, formal syntax, 1-8
COPY macro, 14-31
Correction
    algorithm, 3-36
    code, 3-13, 3-28
COS dataset storage, 2-6
CPB@ (Command Parameter Block), 5-8
CPI@ (Input channel table), 2-55
CPO@ (Output channel table), 2-56
CPW (see Channel Program Word)
CRAY X-MP Computer System, 1-1
CRAYIO, 8-4
CREATE
    function, 2-24
    trace event parameter, 15-15
Create
    activity in another IOP, 15-13
    new activity in the system, 15-15
Creating
    demon activities, 2-8
    overlay table, 2-9
CRT output character, 15-16
CRTOUT trace event parameter, 15-9, 15-16
CRW (see Channel Response Word)
CUT@ (Control Unit Tables), 5-8, 5-15
CV$, 6-12
CXT, 10-12
Cylinder select process (DCU-5), 3-34

D$*name* pointer, 2-8
D2elog trace event parameter, 15-9, 15-16
D2err trace event parameter, 15-9, 15-15
D2micr trace event parameter, 15-9, 15-16
D2seek trace event parameter, 15-9, 15-15
D2stio trace event parameter, 15-9, 15-15
D4*xxx*/D3*xxx* overlays 3-32
D4-*nnn* trace event parameter, 15-11,
  15-20 to 15-21
D4DEM overlay, 3-22, 3-29
D-packet, 4-5
DACT Kernel subroutine, 2-9
DAL (Disk Activity Link), 2-1, 2-19
    entry (DA@@LE), 2-54
    header (DA@@LH), 2-54
    pool, 2-13
Data
    access macros, 14-19
    address word, 5-22
    cache, 3-2
    Chain flag (CPW@CC), 5-11
    definition macros, 14-16
    errors, 3-15
    formats, 4-4
       Interchange format, 4-4
       List I/O format, 4-4
       Transparent format, 4-4
    formatting tape, 1-2
    handler for User Channel shell, 9-7
    movement, 1-1
    streams, 3-1
    Stream Control Table (DSC), 4-1, 4-3
    streaming, 3-2
       between mainframe and tape device,
         4-1, 4-3
    transfer, 5-22
       chained, 5-23
       command, 5-23
       Request (DTR), 3-23
Dataset
    Buffer Memory resident, 2-6

Dataset (continued)
staging, 1-2, 6-2
storage, COS, 2-6
DB@, 3-9
DBT@ (Device Bank Tables), 5-9
DBUD subroutine, 3-5
DCB (Disk Control Block), 3-6, 3-9, 3-22
done queue, 3-3
DCU-4, 3-1
controlling software, 3-2
disk error recovery, 3-13
read-ahead, 3-9
read request stepflow, 3-6
software overlays, 3-2
tables and packet structure, 3-4
write request stepflow, 3-4
disk controller
controlling software, 3-2
error recovery, 3-13
error message, 3-21
read ahead, 3-9
software overlays, 3-2
tables and packet structure, 3-4
DCU-5, 3-11, 15-11, 15-20
controlling software, 3-21
software components, 3-22
tables and packet structure, 3-22
resource management, 3-23
read request stepflow, 3-24
write request stepflow, 3-25
read ahead, 3-26
write behind, 3-27
spiral formatting, 3-28
on-line disk diagnostic requests, 3-28
disk error recovery, 3-29
error message, 3-37
error reporting, 3-39
striped disk groups, 3-40
disk controller, 3-21
controlling software, 3-21
driver tables and packets, 3-22
error
message, 3-37
recovery, 3-29
retry, 11-17
retry limits, 3-30
head select, 15-20
read
ahead and write behind, 3-26
request stepflow, 3-24
resource management, 3-23
sector I/O, 15-20
seek routine, 15-20
software components, 3-22
write
behind, 3-27
request stepflow, 3-25
DD-19 and DD-29 disk
controlling software, 3-2
error recovery, 3-13
error message, 3-21
read ahead, 3-9
software overlays, 3-2
tables and packet structure, 3-4
DD-39, DD-40, and DD-49 disk
controlling software, 3-21
driver tables and packets, 3-22
error
message, 3-37
recovery, 3-29
retry, 15-21
retry limits, 3-30
head select, 15-20
read
ahead and write behind, 3-26
request stepflow, 3-24
resource management, 3-23
sector I/O, 15-20
seek routine, 15-20
software components, 3-22
write
behind, 3-27
request stepflow, 3-25
DD49, 3-21
disk interrupt handler, 3-29
DE sequence check, 3-38
Deactivate activity until popped, 15-13
Deadstart
and restart parameter files, 1-5
I/O Processor, 2-51
Kernel operation at, 2-1
MIOP role, 1-5
overlay space, 2-9
package, 1-1, 2-3
Deallocate memory segment from pool (RELMEM), 2-44
Deallocation and allocation of memory, 2-12
Debug
displays, 6-8
mode for HSX channel software, 12-5
package registers, 1-6
DEBUGGER deck, 15-22
Debugger, 15-22
interactive, 15-1
DEBUGTOO deck, 15-22
Decks, common, 14-2
Delete breakpoint command, 15-29
Demon activities, 2-8, 2-12
Destination ID (DA@DID), 2-55
Device
address/mode word, 5-22
Bank Tables (DBT@), 5-3
configuration (CON$DEV), 5-14
message format, 5-15
Not-ready flag (BDV@NR), 5-19
ordinal (TQ@DVN), 4-5
Parameter Table (DPT), 3-23
table for each configured device, pointer to, 5-7
time-out, 5-18
Diagnostic
on-line system, B-1
request processing, 3-13, 3-28
scratch area, 3-1
system files, 3-1
DIOP responsibilities, 1-5

Disable disk error messages requiring a response, 3-37
Disk
    Activity Link (DAL) pool, 2-13, 2-19
    allocation, 3-1
    buffers, 3-2
    cache, 3-2, 3-26
    control, 3-1
    Control Block (DCB), 3-6, 3-9, 3-22
        done queue, 3-3
    controller, 3-1
        software, DD-19/DD-29, 3-2
        software, DD-39/DD-49, 3-21
    Demon read-ahead
        input/output, 3-12
        process, 3-11
    devices, 3-1
    drivers, 3-1
    driver tables and packets
        DCU-4, 3-4
        DCU-5, 3-22
    driving subroutines (DD-19/DD-29), 3-4
    ERRECK disk retries, 15-16
    error
        handler, 15-15
        information in DAL, 3-20
        logging, 15-16
        message, DD-19/DD-29, 3-20
        message, DD-49/DD-39, 3-37
        recovery, DD-19/DD-29, 3-13
        recovery, DD-39/DD-49, 3-29
    groups, striped, 3-40
    I/O requests, concurrent, 3-1
    I/O, kernel internal, 3-45
    input/output, 3-1
    input/output (I/O) software, 1-2
    interlock, 3-14, 3-16
    interrupt
        answering subroutine (DD-19/DD-29), 3-4
        handler (DD-39/DD-49), 3-29
    queues, local handling of, 3-8
    read ahead control, 3-9
    read request
        stepflow for DD-19/DD-29, 3-6
        stepflow for DD-39/DD-49, 3-24
    read/write processor, 15-15
    requests, 2-6
        processing, 3-2
    Request Packet DAL, 3-22
    seek routine, 15-15
    software validation, 3-2
    storage units (DSUs), maximum, 3-1
    write (DCU-4), 3-13
    write request
        stepflow for DD-19/DD-29, 3-4
        stepflow for DD-39/DD-49, 3-25
DISK overlay, 3-2, 3-3
DISKIO, 3-45
DISP01 routine, 6-10
Display
    accumulator command, 15-24
    B register command, 15-24
    breakpoint command, 15-28
    buffer memory command, 15-30
    carry register command, 15-25
    channel status command, 15-25
    facilities, 6-1
    exit stack command, 15-26
    high-speed channel command, 15-31
    Local Memory command, 15-27
    operand register command, 15-26
    P register command, 15-28
    parameters global symbol, 6-33
DISPLAY task, 6-8
    interaction areas, 6-8 thru 6-10
DK-ERR trace event parameter, 15-9, 15-15
DK-IO trace event parameter, 15-9, 15-15
DK-LOG trace event parameter, 15-9, 15-16
DK-MM trace event parameter, 15-9, 15-16
DK@, 3-22
DKDMP overlay, 15-36
DKIOEX overlay, 3-44
DL@, 3-22
Done queue (DB@DNQ), 3-11
DP@, 3-23
DPT (Device Parameter Table), 3-23
DQTIME, 2-51, 3-18
Driver
    block multiplexer, 4-1
    BMX channel, 5-1
    BMX subsystem channel, 5-21
    call, trace event parameter, 15-21
    disk, 3-1
    dummy, 9-10
    HSX channel, 12-1
    installation, 9-10
    request (CR$DRV), 9-3
    responses, 9-9
    subroutine, UCSHL, 9-7
    User Channel, 9-1
DROP command flow, 6-15
DSC, 4-1, 4-3
DSCGET routine
    description, 4-3
    in free requests, 4-59
DSUs (see Disk storage units)
DTR (Data Transfer Request), 3-23
Dummy drivers, 9-10
Dump
    analysis, A-1
    format, C-7

E-DALs, 3-11
EC$, 2-22
ECL technology and Local Memory, 2-2
EDECR macro, 14-10
EGET macro, 14-9
EINCR macro, 14-9
EITB (Interrupt Jump Table), 5-26
$ELSE macro, 14-12
$ELSEIF macro, 14-12

End
    processing, signal, 2-46
    read requests, tape, 4-37 to 4-40
    Request-in (KIC$ER), 5-28
END command, 6-1
    format, interactive, 8-3
    stepflow, 8-5
End-of-data tape request, 4-37
End-of-File tape request, 4-37
End-of-Record tape request, 4-37
End read tape requests, 4-37 to 4-40
ENDCONC overlay description, 7-8
$ENDTIL macro, 14-13
Entrance to Kernel function processor, 15-12
EOF interactive command, 8-8
EPUT macro, 14-9
Equipment check, 4-62
ERRDMP command, 2-59
ERRECK overlay, 3-2, 3-4, 3-13
Error
    channel processing (IOS serial no. 21 and below), 2-57
    code global symbol, 6-33
    conditions
        DD-19/DD-29, 3-14
        interlock, 3-17
        miscellaneous, 3-17
        in Local Memory, 2-2
    disk recovery
        DD-19/DD-29 disk, 3-13
        DD-39/DD-49 disk, 3-29
        reporting (DCU-5), 3-39
        retry limits (DCU-5), 3-30
        retry (DCU-5), 15-21
        statuses
            DCU-4, 3-20
            DCU-5, 3-39
        summary (DCU-4), 3-18
    display, tape, 4-64
    handler, disk, 15-15
    log, 2-58
    logging
        disk, 15-16
        IOS serial no. 21 and up, 2-59
    message
        DCU-4, 3-21
        DCU-5, 3-37
    messages requiring a response, disable disk, 3-37
    multiplex, 2-59
    NSC activity recovery, 10-10
    procedures, HSX, 12-5 to 12-8
    processing, IOS, 2-57
    reporting (DCU-5), 3-39
    status, 3-20
    tape recovery processing, 4-4, 4-60 to 4-64
ERROR, 2-59
errpt utility, 2-59
Event timer, system, 2-50
Evtout trace event parameter, 15-11, 15-19
Execution control macros, 14-11
Exit from
    activity dispatching, 15-12
    common interrupt handler, 15-12
Exit stack access and macros, 14-8
Expander
    channel interrupt, 15-12
    device control tables, 6-8
EXSGET macro, 14-10
EXSPUT macro, 14-10
EXTRACT utility, 3-39

F-packets, 9-1
    requests, 9-5
FC$BKFIL, tape positioning request, 4-42 to 4-45
    processing of, 4-42
FC$BKSPC, tape positioning request, 4-42 to 4-45
    processing of, 4-42
FC$CHNGE, tape configuration change request, 4-6
    processing of, 4-7
FC$DSP, tape load display request, 4-45 to 4-48
    processing of, 4-46
FC$EODR, tape end-of-data read request, 4-37
    processing of, 4-38
FC$EOFR, tape end-of-file read request, 4-37
    processing of, 4-38
FC$EORR, end-of-record read request, 4-37
    processing of, 4-38
FC$FREE, tape free request, 4-57 to 4-59
    processing of, 4-58
FC$FWFIL, tape positioning request, 4-42 to 4-45
    processing of, 4-42
FC$FWSPC, tape positioning request, 4-42 to 4-45
    processing of, 4-42
FC$MOUNT, tape mount request, 4-6 to 4-8
    processing of, 4-8
FC$NOOP, tape no-op request, 4-40 to 4-41
    processing of, 4-40
FC$READ, tape read request, 4-9 to 4-23
    processing of, 4-9
FC$REWND, tape rewind request, 4-51 to 4-54
    processing of, 4-51
FC$RMNT, remount request, 4-48 to 4-50
    processing of, 4-49, 4-50
FC$RWND1, tape rewind request, 4-51 to 4-54
    processing of, 4-51
FC$RWND2, tape rewind request, 4-51 to 4-54
    processing of, 4-51
FC$UNLD, tape unload request, 4-54 to 4-57
    processing of, 4-55
FC$UNLD1, tape unload request, 4-54 to 4-57
    processing of, 4-55
FC$UNLD2, tape unload request, 4-54 to 4-57
    processing of, 4-55
FC$WRITE, tape write request, 4-23 to 4-36
    processing of, 4-24

FEI logical path
activity, 11-1
initialization, 11-1
termination, 11-1
driver, 1-2
overlay connections, 11-2
overlays
ADEM, 11-2
FNSC, 11-3
FEIR, 11-3
FEIW, 11-3
FEIWMSG, 11-3
status, 15-22
FEI-3, see VMEbus
FEIMSG overlay
FEI logical path, 11-3
VMEbus, 13-4
FEIR
overlay, 11-3,
trace event parameter, 15-12, 15-22
FEIW
overlay, 11-3,
trace event parameter, 15-12, 11-22
FIELD macro, 14-16
FIND function, 2-26
FIRECODE, 3-15
Fixed-size
Local Memory buffer, 2-22
pool, 2-12
FLDADD macro, 14-25
FLDSUB, 14-25
FLUSH
function, 2-26
Kernel call, 2-3
FNSC overlay
FEI logical path, 11-3
NSC, 10-6
VMEbus, 13-2
Formatting
disk spiral, 3-28
tape data, 1-2
Forward space file and record, 4-42 to 4-45
FRAME, 15-2
Free
memory, 2-1
memory chain, 2-12
tape request (FC$FREE), 4-57 to 4-59
FRESTACK macro, 6-4
Front end
computers, 1-2
read/write operation to, 2-21
Front-end concentrator, section 7
CONC overlay description, 7-2
CONCERR overlay description, 7-7
CONCID overlay description, 7-7
CONCIO activity description, 7-2
ENDCONC overlay description, 7-8
Function descriptions, 2-16
Functions
A130OI, 2-21
ALERT, 2-16
ASLEEP, 2-18
AWAKE, 2-19
Functions (continued)
BGET, 2-22
BRET, 2-23
CALL, 2-23
CREATE, 2-24
FIND, 2-26
FLUSH, 2-26
GETDAL, 2-27
GETMEM, 2-27
GIVEUP, 2-28
GOTO, 2-29
HSPR, 2-30
HSPW, 2-32
MGET, 2-33
MOSR, 2-34
MOSW, 2-35
MPUT, 2-36
MSG, 2-37
MSGR, 2-37
OUTCALL, 2-38
OUTPUT, 2-39
PAUSE, 2-40
POLL, 2-40
POP, 2-41
PUSH, 2-42
RECEIVE, 2-43
RELDAL, 2-43
RELMEM, 2-44
RESPOND, 2-44
RETURN, 2-45
SEND, 2-45
TERM, 2-46
TPUSH, 2-46
TRANSFER, 2-47

G-packet, 4-5
General service functions, 2-12
GET macro, 14-20
GETDAL function, 2-27
GETL subroutine, ISP, D-4
GETMEM
function, 2-27
trace event parameter, 15-14
GETSTACK macro, 6-4
GIVEUP function, 2-28
Global
register, 6-2, 6-4, 7-2
symbols, 6-33
$GOTO macro, 14-14
GOTO function, 2-29

H-packets, 12-1
HALT (PUNT) codes, Kernel, 15-33
Halt I/O (RQ$HIO), 5-21
Handling of disk queues, 3-8
Hardware specifications, 1-2
HCOM demon, 12-3
Head select
DD-39 and DD-49, 15-20
-LMA select-read and write
process, 3-29, 3-35, 3-36

High-speed External Communications channel,
see HSX channel interface
History trace, 15-1, 15-4
information buffer, 2-7
sample output, 15-6
HSF$CLOS, 12-3
HSF$CNTL, 12-2
HSF$OPEN, 12-2
HSF$READ, 12-2
HSF$WRIT, 12-2
HSPR function, 2-30
HSPW function, 2-32
HSS$RECI, 12-3, 12-9, 12-10
HSS$SET, 12-2
HSS$SNDI, 12-3, 12-9, 12-10
HST$ errors, 12-6 to 12-8
HSX channel interface, section 12
buffering, 12-4
channel requests, 12-1 to 12-3
debug mode, 12-5
driver architecture, 12-3 to 12-5
error procedures, 12-5to 12-8
interrupt handler, 12-4
overlays, 12-5
HSXI interrupt handler, 12-4
HSXO interrupt handler, 12-4

I-HAND trace event parameter, 15-9, 15-12, 15-13
IACMD overlay, 8-7
IACON overlay, 8-6
IACON1 overlay, 8-7
IAFUNC overlay, 8-4
IAIOP overlay, 8-1
IAIOP1 overlay, 8-4
IAMSG overlay, 8-5
IAOUT overlay, 8-8
IBM-compatible devices, connection, 5-1
IBMX (BMX interrupt handler), 5-26
advance data, 5-27
continue request-in, 5-28
end request-in, 5-28
immediate return, 5-27
routine, 5-1
start request-in, 5-27
ICOM
overlay, 3-22, 3-24
trace event parameter, 15-11, 15-20
Icomsg trace event parameter, 15-11, 15-20
ID
-based table entries, 7-7
errors, 3-16
IDKTOUT routine, 3-18
$IF macro, 14-11
Immediate return (KIC$IR), 5-27
Independent activities, 2-1
Initialization
FEI logical path activity, 11-1
MIOP-mainframe communication, 2-54
NSC activity , 10-1
Initiate
I/O to A130 NSC device, 15-13
output to mainframe through 6 Mbyte channel, 15-13
Input
A-A channel interrupt handlers, 15-12
channel
from the mainframe, 2-55
from mainframe interrupt, 15-13
table (CPI@), 2-55
errors, HSX, 12-6 to 12-7
message packet disposition, 2-56
/output
disk, 3-1
operations, 2-13
stream count (IST), 6-2
INSTACK macro, 6-4
Instruction stack, 1-2
Integrated Support Processor (ISP) channel driver (ISPDRV), D-1
Interactive
concentrator
overlays, 8-1
software, structure of, 8-2
console
overlays, 8-6
software, structure of, 8-2
debugger, 15-1
Reply message, 8-5
station, 1-2, 8-1
Interchange format, 4-4
Intercommunication function codes, I/O Processor, 2-53
Interface
block multiplexer channel, 5-1
routines, block multiplexer channel, 5-12
shell and driver, 9-7
software, block multiplexer channel, 1-2
Interlock
error conditions, 3-17
status, 3-16
Interrupt, 2-1
answering, error channel, 2-58
expander channel, 15-12
handler
DCU-5, 3-26
real-time clock, 2-49
HSX channel, 12-4
input A-A channel, 15-12
overlay registers, 1-6
input channel from mainframe, 15-13
Jump Table (EITB), 5-26
mode, 5-23
occurs due to read ahead, 3-10
pending, 5-25
processing, 2-11, 9-10
Interval counter (%MSEC), 2-49
Intervention required recovery, 4-62
Introduction to disk controlling software, 3-1
INTRPT trace event parameter, 15-8, 15-12

I/O
    buffers, 2-1
       allocation, 2-1
       pool, 2-12
    disk input/output software, 1-1
    request
       for tape, 4-5 to 4-59
       from mainframe, 4-2
       processing, ISP, D-2
       received during read ahead, 3-11
    software, User Channel, 1-2
    time-out, 3-18
    User Channel, 9-1
I/O Processor (IOP)
    central processor queueing and activity
      dispatching, 2-12
    communication, 2-52
    computation section, 1-2
    deadstart, 2-51
    description, 1-2
    intercommunication function codes, 2-53
    message handling, 3-3
I/O stream control tables, 6-8
I/O Subsystem, 1-1
    Block Mux (BMX) subsystem overview, 5-1
    confidence utilities, B-1
    configuration, 1-3
    debugging, 15-22
    editor, 1-5
    error processing, 2-57
    history trace, 15-4
    mainframe communication, 2-54
    model B, 1-3
    model C, 1-3
    operating system, 1-1
    responsibilities, 1-3
    real-time clock, 2-49
    software, parts of, 1-1
    station, section 6
       storage, 6-2
       tasks, 6-1
       task flow and interaction, 6-5
         CLI task, 6-11
         console output, 6-34
         DISPLAY task, 6-8
         global symbols, 6-33
         KEYBD task, 6-6
         POST overlay, 6-32
         PROTOCOL task, 6-16
         screen image, 6-34
         STAGEIN task, 6-25
         STAGEOUT task, 6-27
         station initialization, 6-5
         STIO overlay, 6-29
IOP (see I/O Processor)
IOPPL, 14-1
IOS (see I/O Subsystem)
ISFIELD macro, 14-17
ISP channel driver, D-1
ISROSR, 6-31
Issue a function on a channel command, 15-25
Issue I/O between Buffer Memory and Target
  Memory, 15-19
K-CALL trace event parameter, 15-8, 15-12
K-FNCT trace event paramter, 15-9, 15-13
  to 15-15
K-func trace event parameter, 15-9, 15-13
  to 15-15
Kernel
    active calls, 15-4
    console, send message to, 2-37
    definition 2-1
    description, 1-1
    error logging table, 6-8
    function processor, entrance to, 15-8
    functions, 2-1
    HALT (PUNT) codes, 15-33
    internal disk I/O, 3-45
    operation at deadstart, 2-1
    request destination ID, 2-56
    service calls, 6-3, 6-4
    service request I/O functions, 2-6
    service requests, 2-12
    storage areas, 2-6
    subroutine, DACT, 2-9
KEYBD task, 6-6
    interaction areas, 6-7
KIC$, 5-22, 5-24, 5-25, 5-27, 5-28
    KIC$IR (immediate return), 5-27

LB@, 3-22
LCP
    descriptors global symbol, 6-33
    requests, 7-7
Ldovly trace event parameter, 15-9, 15-15
List of BDV tables by ordinal number, 5-8
List I/O format, 4-4
LISTO overlay, 15-1, 15-34
LISTP overlay, 15-33
Load display tape request, 4-45 to 4-48
LOAD macro, 14-20
Local buffer entry, 3-22
Local handling of disk queues, 3-8
Local Memory
    allocate, 2-27
    buffer control, 3-23
    buffer, fixed-size, 2-22
    chains, 2-12
    data transfer commands, 5-10
    definition, 1-7
    error correction in, 2-2
    management, 3-24
    refresh, 2-3
    scrubbing, 2-2
    stack area, 6-4
    structure, 2-2
    usage, 2-1
LOCK macro, 6-3
LOG interactive command, 8-1, 8-4
Logical ordinal number, 5-7
Logical to physical address mapping,
  striped disk group, 3-41
LOGOFF command, 6-1
LOGOFF interactive command, 8-3, 8-5, 8-8
LOGON interactive command, 8-8

Lost data errors, 3-14, 3-16

Macro
    data access, 14-19
    data definition, 14-16
    execution control, 14-11
    exit stack, 14-8
    general information 14-4
    memory, 14-31
    overlay and register definition, 14-26
    program library (PL) and, 14-1
    service request, 2-13
    summary, 14-5 to 14-8
Macros
    ADDRESS, 14-19
    CLEAR, 14-31
    COPY, 14-31
    EDECR, 14-10
    EGET, 14-9
    EINCR, 14-9
    $ELSE, 14-12
    $ELSEIF, 14-12
    $ENDTIL, 14-13
    EPUT, 14-9
    EXSGET, 14-10
    EXSPUT, 14-10
    FIELD, 14-11
    FLDADD, 14-25
    FLDSUB, 14-25
    GET, 14-20
    $GOTO, 14-14
    $IF, 14-11
    ISFIELD, 14-17
    LOAD, 14-20
    OVERLAY, 14-26
    $PUNTIF, 14-15
    PUT, 14-22
    REGDEFS, 14-27
    REGISTER, 14-28
    RETREG, 14-29
    RGET, 14-22
    RPUT, 14-23
    RSTORE, 14-23
    SIGNAL, 9-7
    STORE, 14-22
    TABLE, 14-18
    $UNTIL, 14-13
    WATCH, 9-7
Main loop, ISP, D-1
Mainframe channels, 2-55
Maintenance computer, 2-57
Manual intervention messages, 3-37
Master DAL, 3-4
Master I/O Processor, 6-1
    responsibilities, 1-5
    mainframe communication channel, 2-54
MCB (Buffer Memory Control Block), 3-23
MEM-IO trace event parameter, 15-11, 15-19 to 15-20
MEM@, 3-23
MEMIO Queue Table, 3-23
Memio trace event parameter, 15-11, 15-19 to 15-20
Memory
    allocating, 2-1
    and deallocation, 2-12
    error correction, 2-2
    errors, soft, 2-2
    free, 2-1
    macros, 14-31
    modification of, 15-22
    search list, 2-10
    type (defined by $APTEXT), 2-6
Message
    areas, 2-5
    formats, 5-14
    handler, User Channel, 9-5
    handling, IOP, 3-3
    Interactive Reply, 8-5
    packet
        allow overlay to send, 2-45
        send B or S type, 2-40
    start, 8-6
Messages, BMX, 5-14 to 5-15
MGET function, 2-33
MGET trace event parameter, 15-14
MIOP (see Master I/O Processor)
Miscellaneous error conditions, 3-18
Modification of
    memory, 15-22
    registers, 15-22
Modification, program library, 14-1
MOSR function, 2-34
MOSW function, 2-35
Mount request (FC$MOUNT), 4-6 to 4-8
    processing flow for, 4-8
Move data between Buffer Memory and Central Memory, 2-47
MPUT function, 2-36
MPUT trace event parameter, 15-14
MSG function, 2-37
MSGR function, 2-37
Multiple-path to multiple-ban linkage, 5-3

N-packets, 10-3, 10-4, 13-1
NIDEND (NSC) overlay, 10-9
NIO Table, 10-2
Nondata transfer commands, 5-9
No-op tape request (FC$NOOP), 4-40 to 4-41
Not capable recovery subroutine, 4-63
NSC A130 adapter, 2-21
NSC
    activity, 10-1
        channel/ID ordinal description, 10-12
        error recovery, 10-10
        initialization, 10-1
        overlays, 10-6
        termination, 10-11
    error recovery, 10-11
        SCP protocol, 10-11
        protocol independent interface, 10-12
    HYPERchannel, 1-2, 10-1
    logical path status, 15-12, 15-22
    messages, 10-9

NSC overlay, 10-9
NSCEND overlay, 10-9
NSCID overlay
    NSC HYPERchannel, 10-9
    VMEbus, 13-5
NSCIO activity, 10-9
    idle loop, 10-2
    read sequence (protocol-indendent
      interface), 10-3
    SCP interface logon sequence, 10-4
    write sequence (protocol-indendent
      interface), 10-3
NSCIO overlay, 10-9
NSCIO trace event parameter, 15-13
NSCMSG overlay, 10-9
NSCRW trace event parameter, 15-12, 15-22
NSCRW overlay
    NSC HYPERchannel, 10-10
    VMEbus, 13-4
NUM, SUMMARY utility, 15-3

O$$OVL, 14-27
Ochar trace event parameter, 15-9, 15-16
Offset algorithm, 3-35
On-line
    access, 15-1
    diagnostic requests
       DCU-4, 3-13
       DCU-5, 3-28
    system diagnostics, B-1
    tape device (thru Block Multiplexer)
      test, B-3
    TRACE commands, 15-4
On-line system diagnostics, appendix B
    CHNTEST, B-2
    CPTEST, B-3
    ECHOCP, B-4
    HSPTEST, B-5
    MOSTEST, B-7
    SSDTEST, B-9
    STOP, B-11
    XDK, B-11
    XMT, B-12
    XPR, B-13
Open
    HSX channel (HSF$OPEN), 12-2
    processing, ISP, D-1
    request (CR$OPN), 9-1
    subroutine, UCSHL, 9-5
Operand register
    assignments, 1-6
    %B, 15-6
    %EX, 2-13
Operating system, IOS, 1-1
Operational description, SYSDUMP, C-1
Operator
    commands, 1-1, 6-1
    displays, 6-8
Ordinal number list of BDV tables by, 5-7
Organization, Buffer Memory, 2-4
OUTCALL function, 2-38

Output
    channel table (CPO@), 2-56
    channel to the mainframe, 2-56
    errors, HSX, 12-7 to 12-8
OUTPUT function, 2-39
Output stream count (OST), 6-2
Overlay
    activate, 2-29
    adding, 14-2
    allocation, user, 2-33
    areas, 2-1
    call, 2-38
    data areas, 2-10
    format, 2-11
    index, 2-9
    load, 2-10
    loading, 15-15
    general information, 2-9
    groupings of, 14-3
    interactive console, 8-6
    interrupt handling, 1-6
    registers, 1-6
       definition, 14-26
         macros, 14-26
    release all, 2-26
    Table, creation, 2-9
OVERLAY macro, 14-26
Overlays
    FEI logical path activity, 11-2
    NSC activity, 10-6
    ACOM, 3-3
    ADEM, 10-6, 11-1, 13-2
    AMAP, 2-1, 2-10, 9-10
    BMXCON, 5-1
    BMXCPU, 4-15, 5-1
    BMXDEM, 5-1
    BMXSIO, 5-1
    CDEM, 3-3
    D4DEM, 3-22
    DD49, 3-22
    DISK demon, 3-6
    DISK, 3-3
    DKDMP, 15-36
    ERRECK, 3-4
    FEIMSG, 11-3
    FEIR, 11-3
    FEIW, 11-3
    FNSC, 11-3, 13-2
    IACMD, 8-7
    IACON, 8-6
    IACON1, 8-7
    IAFUNC, 8-4
    IAIOP, 8-1
    IAIOP1, 8-4
    IAMSG, 8-5
    IAOUT, 8-8
    ICOM, 3-22
    LISTO, 15-1, 15-34
    LISTP, 15-33
    NIDEND, 10-9
    NSC, 10-9
    NSCEND, 10-9
    NSCID, 10-9, 13-5

Overlays (continued)
NSCMSG, 10-9
NSC, 10-9
NSCRW, 10-10, 13-4
OVLNUM, 9-10
PATCH, 15-1, 15-32
REPORT, 3-20
SCPIO, 10-10, 13-5
TERMNSC, 13-5
TERMVME, 13-5
TERNSC, 10-10
TRANSFR, 3-22
VME, 13-4
VMEND, 13-4
VMERD, 13-4
VMEWT, 13-4
Overtemp operator message, 3-38
OVL-LD trace event parameter, 15-9, 15-15
OVLNUM overlay, 9-10, 10-20

Parameter
descriptors global symbol, 3-34
files, deadstart/restart, 1-5
table, trace event, 15-12
Parcel, 1-7
PATCH overlay, 15-1, 15-32
Path, 5-3
PAUSE function, 2-40
Peripheral Expander
confidence tests, B-11 thru B-13
disk unit, B-11
tape unit, B-12
printer unit, B-13
tape/disk, deadstart, 2-51
PL (Program library), 14-1
Pointer
to channel/ device tables for each configured channel, 5-7
POLL
command, 8-3
function, 2-40
trace event parameter, 15-14
Pool
deallocate memory segment from, 2-44
disk Activity Link (DAL), 2-13
fixed-size, 2-12
I/O buffer, 2-12
POP
function, 2-41
trace event parameter, 15-13
Popcell, 2-18
Popping/pushing SMODs, 2-7
Position tape requests, 4-42 to 4-45
POST overlay, 6-32
Priority
demon activity, 2-12
scheme, 2-12
Processing flow for tape requests
configuration change request, 4-7
end read request, 4-38
free request, 4-13
load display request, 4-46
Processing flow for tape requests
mount request, 4-8
no-op request, 4-40
positioning request, 4-31
read request, 4-9
remount request, 4-49, 4-50
rewind request, 4-51
unload request, 4-55
write request, 4-24
Processing of
channels used by the debugger command, 15-31
tape requests, 4-5 to 4-59
Processor queueing and activity dispatching, IOP, 2-12
Program
exit stack, 1-3
library (PL) and macros, 14-1
Protocol independent interface error recovery, 10-12
PROTOCOL task, 6-16
flow
initialization, 6-19
interaction (main body), 6-20
termination, 6-24
PUNT codes, 15-33
$PUNTIF macro, 14-15
PUSH
function, 2-42
trace event parameter, 15-13
Pushing/popping SMODs, 2-7
PUT macro, 14-22

QTIME routine, 2-51, 3-18, 5-24
Queued input dataset information, 6-17

$R@CLI, 6-7
R/W logic power message, 3-38
RD-10 disk
controlling software, 3-21
driver tables and packets, 3-22
error
message, 3-37
recovery, 3-29
retry, 15-21
retry limits, 3-30
head select, 15-20
read
ahead and write behind, 3-26
request stepflow, 3-24
resource management, 3-23
sector I/O, 15-20
seek routine, 15-20
software components, 3-22
write
behind, 3-27
request stepflow, 3-25
Reactivate
activity, 2-41
pushed activity, 15-13

Read
    ahead, 3-2
        Abort flag, 3-11, 3-26
        and write behind (DCU-5), 3-27
        control, disk, 3-9
        Control Table, 3-9, 3-12
        DCU-4, 3-9
        DCU-5, 3-26
        interrupt occurs due to, 3-10
        I/O request received during, 3-11
        process, disk Demon, 3-11
        sequence, 3-10
        steal, 3-11
        tape, 4-2
    data
        from Buffer Memory to Local Memory, 2-34
        from Target Memory to Local Memory, 2-30
    function (FC$READ), 4-15
    request
        HSX channel, 12-2
        tape, 4-9 to 4-23
        User channel I/O, (CR$RD) 9-2
    request stepflow
        DCU-4, 3-6
        DCU-5, 3-24
    subroutine, UCSHL, 9-6
    tape request (FC$READ), 4-9 to 4-23
Read-Hold request (CR$RDH), 9-2
Read/write operation to front end, 2-21
Real-time clock, I/O Subsystem, 2-49
RECEIVE function, 2-43
Receive information from keyboard, 2-43
Receive messages from other IOPs, 15-16, 15-20
Recovery for data errors on read/write operations, 3-15
Recovery subroutines for tape, 4-62 to 4-64
REGDEFS
    definition, 14-7
    macro, 14-27
Register
    and overlay definition, 14-26
    assignments, 1-6
    R!EH, 2-51
    modification of, 15-1
REGISTER macro, 14-28
RELDAL function, 2-43
Release
    all overlays, 2-26
    device path (RQ$RPTH), 5-21
    memory to free pool, 15-14
    message space & reactivate activity, 2-44
    process, 3-29
Relinquish control, 2-28
RELMEM function, 2-44
RELMEM trace event parameter, 15-14
Remount
    tape request (FC$RMNT), 4-48 to 4-51
    tape request processing flow, 4-49, 4-50
REPORT overlay, 3-20
Request
    packet, 3-2
    processing disk, 3-2
    User Channel, 9-1
Request-in sequence (KIC$ER), 5-25
Resource management (DCU-5), 3-23
RESPOND
    function, 2-44
    trace event parameter, 15-13
Response codes (N-packet), 10-6
RETREG macro, 14-29
Retrieving error log information, 2-59
Return
    Buffer Memory to pool, 2-35, 15-14
    control to previous call, 2-45
    DAL to DAL pool, 2-43
    to caller, 5-20
RETURN function, 2-45
Rewind
    tape request, 4-51 to 4-54
    tape request processing flow, 4-51
RGET macro, 14-22
Routines
    BUFMAN, 4-3
    IBMX, 5-1
    IDKTOUT, 3-18
    QTIME, 3-18, 5-23
    SCRUB, 2-3
RPUT macro, 14-23
RQ$, 5-16, 5-20, 5-21
RSTORE macro, 14-23
RTCQUE (timer queue), 2-51
Run switch operator message, 3-38

SCP, 10-11
    protocol error recovery, 10-11
SCPIO trace event parameter, 15-11, 15-22
SCPIO overlay
    NSC, 10-10
    VMEbus, 13-5
Scratch
    area, diagnostics, 3-1
    registers, 14-4, 14-28
Screen image, 6-34
SCRUB routine, 2-3
SDMP0-SDMP1, C-1
SECDED, 2-2
Sector
    chaining, 3-24, 3-26
    I/O, DD-39/DD-49, 15-20
Security erase, data, 4-63
Seek
    errors, 3-14, 3-16
    routine, DD-39 and DD-49, 15-20
SEEK trace event parameter, 15-9, 15-15
Select-in tag (IT$SLI), 5-13
SEND
    function, 2-45
    Kernel service, 9-5
    trace event parameter, 15-14

Send
B or S type message packet, 2-40
characters to the screen, 2-39
message
response to activity in another IOP, 15-13
to Kernel console, 2-37
to other IOPs, 15-15
receive message from Kernel console, 2-37
Sense bit, tape device, 4-61
Sequence code update, 5-24
Service
functions
general, 2-12
summary of, 2-14
request
kernel, 2-12
macro, 2-13
Set
count register and proceed from breakpoint command, 15-28
breakpoint commands, 15-28
Sh-*nnn* trace event parameter, 15-11, 15-21
Shared memory access, 6-3
Shell
and driver interface, 9-7
architecture, 9-4
buffering, 9-9
requests, 9-8
User Channel, 9-1
Signal end of processing, 2-46
SIGNAL macro, 6-3, 9-7
Skip Data Transfer flag, 5-24
SMOD (storage module) creation, 2-7
SNAP command, 6-34
Soft memory errors, 2-2
Software
overlays
DCU-4, 3-2
DCU-5, 3-21
stack, 2-7
Solid state storage device, 1-1
Specifications, hardware, 1-2
Spindle power/speed, 3-38
Spiral formatting (DCU-5), 3-28
SSD 100-Mbyte channel test, B-9
SSD Memory, 1-1, 1-3, 1-5, 1-6, 1-7
Stack Status flag, 5-24
STAGEIN task, 6-25, 6-27
STAGEOUT task, 6-27
Staging, dataset, 1-2
Start
command sequence (KIC$SC), 5-22
I/O (RQ$SIO), 5-16
message, 8-6
Request-in (KIC$SR), 5-27
STARTIO subroutine, ISP, D-3
STATINIT overlay, 6-2
Station
destination ID, 2-56
interactive, 8-1
IOS, 6-1
initialization, 6-5
Station (continued)
storage, 6-2
tasks, 6-1
STATION command, 6-1, 6-5
Statistics, 2-52
destination ID, 2-56
STATUS
command flow, 6-14
interactive command, 8-8
STIO overlay, 6-29
Storage module (SMOD), 2-7
STORE macro, 14-22
Stream state global symbol, 6-34
Streams, data, 3-1
Striped disk groups (DCU-5), 3-40
Structure of
interactive console software, 8-2
Local Memory, 2-2
Subroutines, tape recovery, 4-62 to 4-64
Subsystem overview, IOS Block Mux (BMX), 5-1
Summary of service functions, 2-14
SUMMARY display, 15-2
SUMMARY utility, 15-1
SYSDUMP, C-1
System
configuration, 1-3
Directory, 2-3
Directory contents, 2-5
event timer, 2-50
Log, 3-39, 4-4

$T@KEY, 6-7
Table
access, 14-4
area contents, 2-1
area, TCB, 4-2
and packet structure
DCU-4, 3-4
DCU-5, 3-22
trace event codes, 15-8
TABLE macro, 14-18
TAPDIS
in load display requests, 4-45
in no-op requests, 4-41
in positioning requests, 4-43
in rewind requests, 4-52
in unload requests, 4-54
Tape, section 4
configuration change request (FC$CHNGE), 4-6
control block, 4-2
end read requests, 4-37 to 4-40
error
data security erase, 4-63
display, 4-64
equipment check, 4-62
ID burst check, 4-64
intervention required, 4-62
load point, 4-63
not capable, 4-63
recovery activities, 4-4
recovery processing, 4-60 to 4-64
response packet, 4-4

Tape, (continued)
Exec, section 4
activity, 4-2
bus-out check, 4-62
command reject, 4-63
conditions, 4-4, 4-62 to 4-64
data check, 4-63
data converter check, 4-63
data overrun, 4-63
free request (FC$FREE), 4-57 to 4-59
load display request (FC$DSP), 4-45 to 4-48
mount request (FC$MOUNT), 4-6 to 4-8
no-op request (FC$NOOP), 4-40 to 4-41
packets, 4-5
interprocessor routing of, 4-5
request, 4-5
response, 4-5
positioning requests, 4-42 to 4-45
read request (FC$READ), 4-9 to 4-23
read-ahead data area, 4-2, 4-3
remount request (FC$RMNT), 4-48 to 4-50
request packets, 4-5
request processing, 4-5
response packets, 4-5
rewind requests, 4-51 to 4-54
sense bits, 4-61
unload requests, 4-54 to 4-57
write-behind data area, 4-3, 4-3
write request (FC$WRITE), 4-23 to 4-36
TAPEIO routine
in read requests, 4-19 to 4-22
in write requests, 4-34 to 4-36
TAPEND routine
in end read requests, 4-38 to 4-40
TAPERR routine, 4-4, 4-60
TAPMOV routine
in load display requests, 4-46
in positioning requests, 4-43 to 4-45
in rewind requests, 4-52 to 4-54
in unload requests, 4-55 to 4-57
Target Memory, 1-7, 2-6, 2-47, 15-10, 15-19
Control Block (TMCB), 2-6
Control Table, 2-6
mapping, 3-44
Processor, 1-7, 2-6, 2-47
type, 2-6, 2-48
Task, 1-7
console output, 6-34
flow and interaction, 6-5
global symbols, 6-33
interaction, 6-3
screen image, 6-34
station initialization, 6-5
trace event parameter, 15-8, 15-12
CLI task, 6-11
DISPLAY task, 6-8
KEYBD task, 6-6
POST overlay, 6-32
PROTOCOL task, 6-16
STAGEIN task, 6-25
STAGEOUT task, 6-27
STIO overlay, 6-29
TASK trace event parameter, 15-8, 15-12
TCART overlay, 4-4, 4-61
TCB (tape control block), 4-2
TDEM1 activity
description, 4-4
in read requests, 4-13 to 4-18
in write requests, 4-27 to 4-33
Td0-dn trace event parameter, 15-10, 15-18
Td0-in trace event parameter, 15-10, 15-18
Tdm-dn trace event parameter, 15-10, 15-18
Tdm-in trace event parameter, 15-10, 15-18
TERM function, 2-46
Termination
FEI logical path activity, 11-1
NSC activity, 10-6
Terminology, 1-7
TERMNSC (NSC) overlay, 10-10, 13-5
TERMVME overlay, 13-5
TERROR overlay, 4-4, 4-60 to 4-61
TEST-I/O command, 5-13
TEX routine
in end read requests, 4-38, 4-40
in free requests, 4-59
in load display requests, 4-45, 4-46, 4-48
in mount requests, 4-8
in no-op requests, 4-41
in positioning requests, 4-43, 4-45
in read requests, 4-19, 4-22, 4-23
in remount requests, 4-48 to 4-50
in rewind requests, 4-52, 4-54
in unload requests, 4-54, 4-55, 4-57
in write requests, 4-33 to 4-34, 4-36
TEX*n*, trace event parameters, 15-10, 15-17 to 15-19
TF$, 15-8
Time-out
channel, 5-24
clock-event, 15-19
device, 5-18
I/O, 3-18
Timer
entry, 2-51
queue (RTCQUE), 2-51
TLOC pointer, 15-4
TM$, 2-6
TM@, 2-6
TMOUT trace event parameter, 15-11, 15-19
TMR@TM, 2-49
Toggle display mode command, 15-27
Tpi-rs trace event parameter, 15-10, 15-18
TPTRL pointer, 15-4
TPUSH
function, 2-46
Kernel call, 2-3
Kernel service request, 9-10
TR@, 3-23
TRACE call, 15-4
Trace event
codes table, 15-8
parameters table, 15-12
Transfer commands, 5-9 to 5-11
TRANSFER function, 2-47
TRANSFR overlay, 3-22

Transparent format, 4-4
Two one-by-one configurations (single path, multiple bank), 5-5
Two-by-one configuration (multiple-path, multiple-bank), 5-6
Two-by-two configuration (multiple path, single bank), 5-4

UC$, 9-8
UC@REQ field, 9-8
UCDRV0 through UCDRV9, 9-10
UCH*n* trace event parameter, 15-11, 15-21
UCRD/UCWRT trace event parameter, 15-21
UCSHL (User Channel shell), 9-5
    close subroutine (UCCLS), 9-6
    driver subroutine (UCDRV), 9-7
    open subroutine (UCOPN), 9-5
    read subroutine (UCRD), 9-6
    trace event parameter, 11-17
    write subroutine (UCWRT), 9-6
UCT (User Channel Table), 9-4
UCXFR, 9-6
UNICOS FEI connection, 11-1
Unit
    check/unit exception, 5-25
    ready operator message, 3-38
    release process, 3-37
    select process, 3-33
UNLOCK macro. 6-3
Unload tape requests, 4-54 to 4-57
$UNTIL macro, 14-3
USCP (UNICOS Station Call Processor), 11-1
User Channel
    configuration, 9-10
    drivers, 9-1
    I/O software, 1-2
    I/O, 9-1
    message handler, 9-5
    requests, 9-1
    shell (UCSHL), 9-1, 9-5
    shell data handler (UCXFR), 9-7
    Table (UCT), 9-4
User overlay allocation, 2-33
Utilities
    EXTRACT, 3-39
    SUMMARY, 15-1

VME overlay, 10-13, 13-4
VMEbus, section 13
    description, 1-2
    interrupt handling, 13-8
    N-packet interface, 13-1
    Overlay connections, 13-1
    Overlays, 13-2 to 13-5
    Read and write requests, 13-5 to 13-7
    SCP protocol, 13-7 to 13-8
VMEND overlay, 10-13, 13-4
VMERD overlay, 13-4
VMEWT overlay, 13-4

Wait I/O (RQ$WIO), 5-20
WAITIO subroutine, ISP, D-3
WATCH macro, 6-3, 9-7
Word, 1-7
Write
    behind
        DCU-4, 3-6, 3-12
        DCU-5, 3-27
        tape, 4-2
    data
        from Local Memory to Target Memory, 2-32
        to Buffer Memory from Local Memory, 2-35
    FC$WRITE, 4-23 to 4-36
    -Hold request (CR$WRTH), 9-3
    interchange format, 4-4
    protect operator message, 3-38
    request
        HSX channel, 12-2
        tape (FC$WRITE), 4-23 to 4-36
        user channel (CR$WRT), 9-2
    request stepflow
        DCU-4, 3-4
        DCU-5, 3-25
    ring status, 4-63
    subroutine, UCSHL, 9-6
    -Write request (CR$WRT2), 9-3

XCBT pointer, 5-7
XCHT pointer, 5-7
XDBT pointer, 5-7
XDEV pointer, 5-7
XDEVMAX entry, 4-5
XIOP
    in tape processing, 4-1
    responsibilities, 1-6

READER'S COMMENT FORM

IOS Software Internal Reference Manual SM-0046 G

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): _____
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name _____
Title _____
Company _____
Telephone _____
Today's Date _____

Address _____
City _____
State/ Country _____
Zip Code _____

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE

READER'S COMMENT FORM

IOS Software Internal Reference Manual SM-0046 G

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): _____
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name _____
Title _____
Company _____
Telephone _____
Today's Date _____

Address _____
City _____
State/ Country _____
Zip Code _____

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE