VAXcluster Disk I/O Internals Manual

Order Number:

March 1988

This Manual describes the Internals of performing MSCP access to disks

AUTHOR: Roy G. Davis

Updated By: Robert A. Premovich

Revision/Update Information:

This manual supersedes the Disk I/O Internals Manual, Version 1.0 Last Update (16-Sep-1992)

Operating System and Version:

VMS Version 5.5

Software Version:

Version 5.5

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

October 1990

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

PREFACE xv

CHAPTER 1 SCA AND SCS CONCEPTS 1–1

1.1 INTRODUCTION 1–1

1.2 MASS STORAGE COMMUNICATIONS 1–1

1.3 INTER-SYSTEM COMMUNICATIONS 1–3

1.4 INFORMATION EXCHANGE 1–3
- 1.4.1 Datagrams 1–3
- 1.4.2 Messages 1–4
- 1.4.3 Block Data Transfers 1–4

1.5 COMMUNICATION MECHANISMS 1–4
- 1.5.1 SCA Ports 1–4
  - 1.5.1.1 Definition • 1–4
  - 1.5.1.2 Port Drivers • 1–5
  - 1.5.1.3 Local Controllers • 1–6
  - 1.5.1.4 Port Descriptors • 1–6
- 1.5.2 SCA Virtual Circuits 1–7
  - 1.5.2.1 Definition • 1–7
  - 1.5.2.2 Virtual Circuit Data Structures • 1–7
  - 1.5.2.3 System Blocks • 1–7
  - 1.5.2.4 Path Blocks • 1–8
  - 1.5.2.5 Virtual Circuit Data Organization • 1–8
- 1.5.3 SCA Connections 1–8
  - 1.5.3.1 Definition • 1–8
  - 1.5.3.2 Connection Descriptors • 1–9
  - 1.5.3.3 Connection Structures • 1–9
- 1.5.4 Communications Mechanisms Example 1–10

1.6 ARCHITECTURAL LAYERS OF SCA 1–11
- 1.6.1 SYSAP Layer 1–12
  - 1.6.1.1 Definition • 1–12
- 1.6.2 SCS Layer 1–12
  - 1.6.2.1 Port Independent SCS Services • 1–12
    - 1.6.2.1.1 Connection Management Services • 1–12
    - 1.6.2.1.2 Directory Services • 1–13
    - 1.6.2.1.3 SCS Process Polling Services • 1–14
    - 1.6.2.1.4 SYSAP Connection Analogy • 1–15
  - 1.6.2.2 Port Dependent SCS Operations • 1–16
- 1.6.3 PPD Layer 1–17
- 1.6.4 PI Layer 1–17

1.7 VMS IMPLEMENTATION OF SCA ARCHITECTURAL LAYERS 1–17
- 1.7.1 DUDRIVER CONNECTs to MSCP Disk Server 1–20
  - 1.7.1.1 Local Node • 1–20
  - 1.7.1.2 Remote Node • 1–20
  - 1.7.1.3 Connection Data Structures • 1–21

1.7.2 **DUDRIVER Sends MSCP Command to MSCP Disk Server** 1–22
1.7.2.1 Buffer Allocation • 1–23
1.7.2.2 Identifying the Receiving Sysap and Connection • 1–23
1.7.3 **MSCP Server Sends END Message to DUDRIVER** 1–23
1.7.3.1 Locating the Connection Associated with an End Message • 1–23
1.7.4 **Response IDs and Command Reference Numbers** 1–24
1.7.4.1 Class Driver Request Packet • 1–24
1.7.4.2 Request Descriptor Table Entries • 1–24
1.7.4.3 MSCP Server End messages • 1–25
1.7.5 **DUDRIVER and Block Data Transfers** 1–26
1.7.5.1 Buffer Descriptors • 1–26
1.7.5.2 Buffer Handles • 1–27
1.7.6 **Concept of Flow Control** 1–28
1.7.6.1 Credit Scheme • 1–28
1.7.6.2 Piggybacking • 1–29
1.7.7 **MSCP Server in a Controller** 1–30
1.7.7.1 Local Server Handles SCA Events Essentially the Same • 1–30

CHAPTER 2 DUDRIVER I/O DATABASE 2–1

2.1 INTRODUCTION 2–1

2.2 DATA STRUCTURES 2–1
2.2.1 **SB - System Block** 2–2
2.2.1.1 Configuration List of System Blocks • 2–3
2.2.1.2 System Blocks and CI Ports • 2–3
2.2.1.3 System Blocks and NI Ports • 2–4
2.2.1.4 System Blocks and Local DSA Controllers • 2–5
2.2.2 **DDB - Device Data Block** 2–5
2.2.2.1 DDBs and Remote DSA Controllers • 2–5
2.2.2.2 DDBs and Remote MSCP-Served Disks • 2–6
2.2.2.3 DDB Chain for Local DSA Disks • 2–7
2.2.2.4 DDB for Boot Device • 2–8
2.2.3 **UCB - Unit Control Block** 2–8
2.2.3.1 Linked Lists of UCBs • 2–8
2.2.3.2 UCBs for DSA and MSCP-Served Disks • 2–9
2.2.4 **CDDB - Class Driver Data Block** 2–11
2.2.4.1 Linkage From UCBs to CDDB for Controller • 2–12
2.2.4.2 Linkage from CDDB to UCBs on that Controller • 2–14
2.2.4.3 Linkage from CDDB to DDBs on that Controller • 2–14
2.2.4.4 Extensions to Disk Class Driver CDDB • 2–15
2.2.5 **CRB - Channel Request Block** 2–16
2.2.6 **Dual-Pathed Disks** 2–20

2.3 DUDRIVER I/O DATABASE INITIALIZATION 2–24
2.3.1 **DUDRIVER's Controller Initialization Routine** 2–24
2.3.1.1 DU_CONTROLLER_INIT • 2–24

2.3.2 **Overview of DUDRIVER's Controller Initialization Routine** 2–28
2.3.2.1 CDDB Creation and Initialization • 2–29
2.3.2.2 MAKE_CONNECTION Establishes a Connection to MSCP Server • 2–29
2.3.2.3 Poll for Disk Units • 2–29
2.3.2.4 Check for Controller Based Shadow Set • 2–29
2.3.2.5 Handling of Secondary Path Discovery • 2–30
2.3.3 **Determine Access Paths Processing** 2–32
2.3.3.1 Determination of Topology of Disk Units • 2–32
2.3.3.2 Access Path Attention Messages • 2–32
2.3.3.3 Setup of Dual Path if Found • 2–32
2.3.3.4 DAP Scheduling • 2–32
2.3.3.4.1 DAPBSY Flag Set in the CDDB if DAP Processing in Progress • 2–33
2.3.3.4.2 DAPBSY Flag Checked for DAP Already in Progress • 2–33
2.3.3.4.3 DAPCOUNT Field used to Determine Frequency of DAP Processing • 2–33
2.3.4 **Attention Messages** 2–34
2.3.4.1 Unit Available Attention Message • 2–34
2.3.4.2 Duplicate Unit Attention Message • 2–34
2.3.4.3 Access Path Attention Message • 2–34
2.3.5 **The CONFIGURE Process** 2–35
2.3.5.1 Configure uses SCS Process Polling to Discover MSCP Servers • 2–35
2.3.5.2 Requesting Polling • 2–36
2.3.5.3 Discovery of MSCP Controllers • 2–36

2.4 **DUDRIVER I/O DATABASE INITIALIZATION ROUTINES** 2–38
2.4.1 **DU_CONTROLLER_INIT** 2–39
2.4.1.1 Routine Process • 2–39
2.4.2 **MAKE_CONNECTION** 2–41
2.4.2.1 Establishing a Connection • 2–41
2.4.3 **DUTU$POLL_FOR_UNITS** 2–43
2.4.3.1 Polling Loop • 2–43
2.4.3.2 Polling for Units Complete • 2–45
2.4.4 **DUTU$NEW_UNIT** 2–45
2.4.4.1 Determines if Unit Already Seen on Controller • 2–46
2.4.4.2 Unit Already Seen On This Controller • 2–46
2.4.4.3 Unit Not Already Seen On This Controller • 2–47
2.4.5 **DUTU$DODAP** 2–48
2.4.5.1 Preparations for Performing DAP Processing • 2–48
2.4.5.2 Issues DAP Commands to Controller • 2–49
2.4.6 **ATTN_MSG** 2–49
2.4.6.1 Unit Available Attention Message • 2–50
2.4.6.2 Duplicate Unit Attention Message • 2–51
2.4.6.3 Access Paths Attention Message • 2–51
2.4.7 **Routines in the CONFIGURE Process** 2–51
2.4.7.1 Polling for MSCP Servers on Other Nodes • 2–51
2.4.7.2 CONFIGURE Notified of Discovery of MSCP$DISK • 2–52

Contents

CHAPTER 3 $QIO SYSTEM SERVICE AND DUDRIVER 3–1

3.1 INTRODUCTION 3–1

3.2 ASSIGNING AN I/O CHANNEL TO A DISK 3–1
3.2.1 Assign System Service 3–1
3.2.2 Channel Control Blocks 3–2
3.2.2.1 Maximum Channel Limit • 3–2
3.2.2.2 Channel Number • 3–2
3.2.2.3 Numerical Representation of Access Mode • 3–3
3.2.3 Volume Set Considerations 3–4
3.2.4 Overview of Steps Taken by SYS$ASSIGN 3–4

3.3 OPENING A FILE 3–5
3.3.1 Window Control Blocks and Mapping a File 3–5
3.3.1.1 Virtual Blocks • 3–5
3.3.1.2 Logical Blocks • 3–6
3.3.1.3 Bad Block Replacement • 3–6
3.3.1.4 Window Control Blocks • 3–6
3.3.2 Mapping Situations Requiring Special Handling 3–8
3.3.2.1 Window Turns • 3–8
3.3.2.2 Bound Volume Sets • 3–8
3.3.2.3 File Fragmentation • 3–9

3.4 DRIVER DATA STRUCTURES, THE IRP, DDT AND FDT 3–9
3.4.1 I/O Request Packet 3–9
3.4.1.1 Class Driver Request Packet • 3–10
3.4.2 Driver Dispatch Table 3–11
3.4.3 Function Decision Table 3–12
3.4.3.1 Valid I/O Function Mask • 3–13
3.4.3.2 Buffered I/O Function Mask • 3–13
3.4.3.3 Applicability and Routine Entries • 3–13

3.5 OVERVIEW OF THE FLOW OF A $QIO 3–14
3.5.1 The Process's Point of View 3–16
3.5.1.1 Queuing the Request to the Driver • 3–16
3.5.1.2 Driver Handles $QIO Request • 3–17
3.5.1.3 AST Notification • 3–17
3.5.1.4 Event Flag Notification • 3–17
3.5.2 What VMS Sees 3–18
3.5.2.1 CI and DSSI Ports • 3–19
3.5.2.2 Local Ports • 3–19
3.5.2.3 NI Ports • 3–19
3.5.2.4 I/O Pre-processing • 3–19
3.5.2.5 DUDRIVER Builds MSCP Command • 3–20
3.5.2.6 Transmission of the Command to the Controller • 3–20
3.5.2.7 End Message Received from Controller • 3–20
3.5.2.8 Class Driver Processes End Message • 3–20
3.5.2.9 I/O Postprocessing and AST Delivery • 3–21

3.6 DETAILS OF THE FLOW OF A $QIO 3–21
3.6.1 Device Independent I/O Pre-processing 3–23
3.6.2 Device and Function Dependent I/O Pre-processing 3–24
3.6.3 Class Driver SCS Resource Allocation 3–26

3.6.4 DUDRIVER Builds MSCP Command 3–28
3.6.5 Transmission of Message by SCS and PPD Layers 3–29
3.6.6 End Message Received by PPD and SCS Layers 3–30
3.6.7 Disk Class Driver Message Input Dispatching Routine 3–31
3.6.8 Class Driver Thread Resumes 3–31
3.6.9 I/O Postprocessing and AST Delivery 3–32

3.7 IMPACT ON $QIO FLOW DUE TO LOCAL DSA CONTROLLER 3–33
3.7.1 Allocating an SCS Message Buffer 3–34
3.7.1.1 Ring Buffer Count Calculation • 3–34
3.7.2 Mapping the IRP 3–37
3.7.2.1 The Case of the UDA50 • 3–37
3.7.2.2 Other DSA Controllers • 3–42
3.7.3 Transmission of SCS Message Buffer Containing MSCP Command 3–43
3.7.3.1 Use of the Command Ring • 3–43
3.7.3.2 Reclaiming Descriptors and Buffers from the Command Ring • 3–46
3.7.4 Receiving MSCP End Message from a Local DSA Controller 3–47
3.7.5 Deallocating the SCS Message Buffer 3–50

CHAPTER 4 DISK CLASS DRIVER ERROR HANDLING AND BUGCHECKS 4–1

4.1 INTRODUCTION 4–1

4.2 DUDRIVER TIMEOUT MECHANISM 4–1
4.2.1 Overview of the Timeout Mechanism 4–2
4.2.2 Detailed Flow of DU$TMR 4–3
4.2.2.1 No Commands active for Controller • 4–3
4.2.2.2 Commands Are Still active for Controller • 4–4

4.3 MSCP END MESSAGES WITH ERROR STATUS CODES 4–6
4.3.1 Detecting File Read/Write Errors and Dispatch 4–10
4.3.2 Errors Returned in End Messages for File Read/Write Requests 4–11
4.3.3 Handling Errors Returned in Read/Write End Messages 4–13
4.3.3.1 Specially Handled Error Conditions • 4–14
4.3.3.1.1 *Invalid Command* Major Status Code • 4–14
4.3.3.1.2 *Host Buffer Access Error* Major Status Code • 4–15
4.3.3.1.3 *Available* Major Status Code • 4–15
4.3.3.1.4 All Other Errors • 4–15
4.3.4 Errors Returned in Other End Messages 4–15
4.3.5 Error Logging and Error Count Incrementing 4–16

4.4 SYNCHRONIZING WITH AN "MSCP SPEAKING" CONTROLLER 4–17
4.4.1 Errors Causing Resynchronization with an MSCP Server 4–17
4.4.2 Overview of Resynchronization Due to Errors 4–18
4.4.3 DU$RE_SYNCH and DU$CONNECT_ERR Detail 4–22

4.5 MOUNT VERIFICATION 4–26
4.5.1 Circumstances Leading to Mount Verification 4–29
4.5.2 Disks Which Qualify for Mount Verification 4–31
4.5.3 Failover of Dual-Pathed Disks 4–32

4.5.4 **Mount Verification Volume Validation** 4–35
4.5.4.1 Perform_Validate Routine • 4–36
4.5.4.2 PACKACK_VOLUME Routine • 4–37
4.5.4.3 VALIDATE_VOLUME Routine • 4–39
4.5.5 **Mount Verification Timeout** 4–39
4.5.6 **Disks Requiring Special Handling** 4–39
4.5.6.1 Foreign Disks • 4–40
4.5.6.2 System Disk and Quorum Disk • 4–40
4.5.7 **Stalling and Unstalling I/O During Mount Verification** 4–40
4.5.8 **Aborting Mount Verification** 4–42
4.5.9 **Mount Verification - The Big Picture** 4–42
4.5.10 **Mount Verification Routines** 4–45
4.5.10.1 DUTU$REVALIDATE • 4–45
4.5.10.2 EXE$MOUNTVER • 4–47
4.5.10.3 PACKACK_VOLUME • 4–54
4.5.10.4 EXE$MNTVERSIO and Handling IO$_PACKACK MVIRP • 4–55
4.5.10.5 Restarting CDRPs • 4–57
4.5.10.5.1 DUTU$RESTART_NEXT_CDRP and DUTU$END_SINGLE_STREAM • 4–57
4.5.10.5.2 Preventing an Infinite Loop • 4–58

4.6 **DUDRIVER BUGCHECKS FOR NON-SHADOWED DISKS** 4–59

CHAPTER 5 THE VMS BASED MSCP SERVER 5–1

5.1 **INTRODUCTION** 5–1

5.2 **MSCP DISK SERVING** 5–1
5.2.1 **Automatic Disk Serving** 5–2
5.2.2 **Selective Disk Serving** 5–2
5.2.3 **Dual Ported Disks** 5–2
5.2.4 **The MSCP Server** 5–3

5.3 **MSCP SERVER DATABASE AND INITIALIZATION** 5–4
5.3.1 **MSCP Server Data Structures** 5–5
5.3.1.1 HRB - Host Request Block • 5–5
5.3.1.2 HQB - Host Queue Block • 5–6
5.3.1.3 UQB - Unit Queue Block • 5–7
5.3.1.4 HULB - Host Unit Load Block • 5–8
5.3.1.5 DSRV - Disk Server Structure • 5–8

5.4 **MSCP UNIT NUMBERS AND IDENTIFIERS** 5–11
5.4.1 **MSCP Media Identification** 5–11
5.4.2 **Unit Identifier** 5–13
5.4.2.1 MSCP Class Number • 5–16
5.4.2.2 MSCP Model Number • 5–16
5.4.2.3 MSCP Unique Device Number • 5–16
5.4.3 **Host Numbers** 5–19
5.4.4 **Transfer Buffers** 5–20
5.4.4.1 Transfer Buffer Allocation • 5–21
5.4.5 **VMS based MSCP server Flow Control** 5–25
5.4.6 **Controller Timeout** 5–25

5.4.7 MSCP Server Initialization Overview 5–25
5.4.8 Loading and Starting the MSCP Server 5–26
5.4.9 Serving Devices 5–27
5.4.10 ACCEPTing an SCS CONNECT From a Remote Host 5–28

5.5 MSCP SERVER LOAD BALANCING 5–30
5.5.1 Static Load Balancing 5–30
5.5.2 Load Monitoring Thread 5–31

5.6 MSCP SERVER'S HANDLING OF READ AND WRITE COMMANDS 5–32
5.6.1 Overview of MSCP Server Handling READ Command 5–35
5.6.2 Overview of MSCP Server Handling WRITE Command 5–39
5.6.3 Command Status 5–40
5.6.4 Details of the Routines for Handling READ and WRITE Commands 5–42
5.6.4.1 MSG_IN - Receiving Command and Server Resource Allocation • 5–42
5.6.4.2 NONSEQB - Verifying that Command Processing may Continue • 5–43
5.6.4.3 READ - Processing MSCP READ Command • 5–44
5.6.4.4 IOC$IOPOST - I/O Postprocessing for READ • 5–46
5.6.4.5 Read Request Resumes Following DO_DISK • 5–46
5.6.4.6 WRITE - Processing MSCP WRITE Command • 5–48
5.6.4.7 IOC$IOPOST - I/O Postprocessing for WRITE • 5–50
5.6.4.8 Write Request Resumes Following DO_DISK • 5–50
5.6.4.9 SEND_END - Send End Message and Cleanup • 5–51

5.7 OTHER CLASSES OF COMMANDS HANDLED BY THE SERVER 5–51
5.7.1 Overview 5–52
5.7.1.1 Immediate Commands • 5–52
5.7.1.2 NonSequential Commands • 5–52
5.7.1.3 Sequential Commands • 5–53
5.7.2 Immediate Class Commands 5–54
5.7.2.1 Routines for Handling Immediate Commands • 5–58
5.7.2.1.1 ABORT • 5–58
5.7.2.1.2 GET_COMMAND_STATUS • 5–60
5.7.2.1.3 GET_UNIT_STATUS • 5–60
5.7.2.1.4 SET_CONTROLLER_CHAR • 5–61
5.7.3 Non-Sequential Non-Buffered Class Commands 5–62
5.7.3.1 Access Command • 5–63
5.7.3.2 Replace Command • 5–63
5.7.3.3 Compare Controller Data and Flush Commands • 5–63
5.7.3.4 Erase Command • 5–63
5.7.4 Routines for Handling Non-Sequential Non-Buffered Commands 5–65
5.7.4.1 ACCESS Routine • 5–65
5.7.4.2 COMP_CTRL_DATA Routine • 5–65
5.7.4.3 ERASE Routine • 5–65
5.7.4.4 FLUSH Routine • 5–67
5.7.4.5 REPLACE Routine • 5–67
5.7.5 Sequential Class Commands 5–67

	5.7.6	**Routines For Handling Sequential Commands**	5–71
		5.7.6.1 AVAILABLE • 5–71	
		5.7.6.2 ONLINE • 5–72	
		5.7.6.3 SET_UNIT_CHR • 5–73	
		5.7.6.4 DET_ACC_PATH • 5–74	
	5.7.7	**Sequential Commands, Nonsequential Commands, and Blocking**	5–74
		5.7.7.1 Basic Scenario • 5–75	
		5.7.7.2 Special Case • 5–81	
5.8	**ERROR HANDLING**		5–82
APPENDIX A	**SYMBOL TABLES AND DATA STRUCTURES**		A–1
A.1	**SDA SYMBOL TABLES**		A–1
A.2	**PUBLIC LIBRARIES**		A–2
A.3	**SDL FILES**		A–4
A.4	**USER CREATED SYMBOL TABLES**		A–6
A.5	**DATA TYPE NAMING CONVENTIONS**		A–7
APPENDIX B	**DATA STRUCTURES**		B–1
B.1	**CCB - CHANNEL CONTROL BLOCK**		B–2
B.2	**CDDB - CLASS DRIVER DATA BLOCK**		B–4
B.3	**CDRP - CLASS DRIVER REQUEST PACKET**		B–11
B.4	**CRB - CHANNEL REQUEST BLOCK**		B–15
B.5	**DDB - DEVICE DATA BLOCK**		B–19
B.6	**DSRV - DISK SERVER STRUCTURE**		B–21
B.7	**HQB - HOST QUEUE BLOCK**		B–29
B.8	**HRB - HOST REQUEST BLOCK**		B–31
B.9	**HULB - HOST UNIT LOAD BLOCK**		B–35
B.10	**IRP - I/O REQUEST PACKET**		B–36
B.11	**SB - SYSTEM BLOCK**		B–43
B.12	**UCB - UNIT CONTROL BLOCK**		B–46
B.13	**UCB ERROR LOG EXTENSION**		B–58
B.14	**UCB DUAL PORT EXTENSION**		B–60
B.15	**UCB DISK EXTENSION**		B–61
B.16	**UCB MSCP EXTENSION**		B–62
B.17	**UCB DUDRIVER EXTENSION**		B–68
B.18	**UQB - UNIT QUEUE BLOCK**		B–70
B.19	**VCB - VOLUME CONTROL BLOCK COMMON DEFINITIONS**		B–74

B.19.1 Volume Control Block fields for Disks B–76

APPENDIX C CROSS REFERENCE C–1

INDEX

FIGURES

1–1 CI Node, Port and Physical Interconnect relationship 1–5
1–2 Port Driver, Port and Physical Interconnect configuration for both the CI and NI model 1–6
1–3 System Block and Path Block linkage 1–8
1–4 MSCP Server to Class Driver Message Flow 1–10
1–5 Telephone System Analogy to Systems Communications Architecture 1–11
1–6 The Architectural Layers of SCA 1–11
1–7 Example of a SYSAP in a Listening State 1–14
1–8 SCS Process Poll Block Linkage 1–15
1–9 SCA Flow as Implemented on VMS 1–19
1–10 The Message Flow of a Disk Class Driver Forming a Connection with an MSCP Server 1–21
1–11 Data Structures for a Formed Connection 1–22
1–12 Fork Process Thread association through the RSPID 1–25
1–13 Layout of the Buffer Descriptor 1–26
1–14 Example of a Buffer Descriptor for a Three Page Transfer 1–27
1–15 The Buffer Handle 1–28
2–1 System Block List 2–3
2–2 DDB Linkage off of the System Block 2–6
2–3 DDB Linkage for a Local Served Disk 2–7
2–4 DDB Linkage Showing Four Disks 2–9
2–5 UCB Extensions for MSCP Served Disk 2–11
2–6 CDDB Linkage Maintained by each UCB 2–13
2–7 CDDB Format and Class Driver Extensions 2–16
2–8 CRB Timeout Linkage 2–17
2–9 CRB Linkage and the General Related Data Structures 2–19
2–10 Data Structures Supporting Secondary Paths 2–21
2–11 Provisions for Secondary Paths Offered by Multiple Servers 2–23
2–12 Configuration of Devices by Sysboot and Init 2–27
2–13 Configuration of Devices by Sysinit and Startup 2–28
2–14 DU_CONTROLLER_INIT flow 2–31
2–15 Attention Message Dispatching 2–35
2–16 Configure Process Polling and Device Configuration 2–37
3–1 Window Control Block Fields for VBN to LBN Translation 3–7
3–2 Window Control Block Mapping 3–7
3–3 IRP/CDRP pair organization 3–11

3–4	FDT layout	3–12
3–5	FDT processing	3–14
3–6	QIO Flow Through the Class Driver	3–22
3–7	Local Port Buffer Initial Layout	3–36
3–8	Vax 11/780 Adapter Configuration	3–38
3–9	Unibus to SBI Mapping	3–39
3–10	Buffer Handle for a UDA buffer	3–41
3–11	Unibus to CMI Mapping	3–42
3–12	Command Ring Format	3–44
3–13	Command Ring Descriptors	3–44
3–14	Local Port SCS Message Buffer Format	3–45
3–15	Response Ring Buffer Pointers	3–49
4–1	CRB Timeout Mechanism and linkage	4–2
4–2	MSCP End Message Status Return Format	4–7
4–3	MSCP Read Request Message Flow	4–8
4–4	MSCP Write Request Message Flow	4–9
4–5	DUDRIVER resynchronization flow	4–21
4–6	Mount Verification Validation Flow	4–28
4–7	Dual Pathed Disk Device Configuration	4–33
4–8	Failover of a Dual Pathed HSC disk	4–34
4–9	Volume Validation Flow	4–36
5–1	MSCP Server Flow	5–4
5–2	HRB fields	5–6
5–3	HRB relationship to the Host Queue Block	5–7
5–4	Disk Server Structure Layout	5–10
5–5	Server Local Unit Number	5–12
5–6	MSCP Unit Number for the VMS based MSCP Server	5–12
5–7	64 bit Unique Identifiers	5–13
5–8	Unit Identifier Format for VMS Based MSCP Servers	5–19
5–9	Host Index Bitfield at DSRV$B_HOSTS in the DSRV	5–20
5–10	UQB's Online Field Bitmap of Hosts Accessing a Specific Unit	5–20
5–11	Free Transfer Buffer Linkage	5–21
5–12	Initial State of the Transfer Buffer	5–22
5–13	Transfer Buffer Allocation and Deallocation	5–23
5–14	General Flow of VMS based MSCP server Reads and Writes	5–34
5–15	Data Structures and Linkage Involved in a Server Receiving a Command	5–36
5–16	Data Structures Involved with MSCP Read and Write Commands	5–38
5–17	General Flow of Immediate Class Commands	5–57
5–18	NonSequential NonBuffered Command Flow	5–64
5–19	General Flow of Sequential Commands	5–70
5–20	Processing NonSequential Commands with No Sequential Commands Issued	5–75
5–21	Sequential Command Received While Processing NonSequential Commands	5–76

5–22	Sequential Command Pending With NonSequential Commands Arriving	5–77
5–23	Currently Executing Commands Have Completed With Commands Queued	5–78
5–24	Sequential Command Begins Execution	5–79
5–25	Sequential Command Executing with NonSequential Commands Arriving	5–80
5–26	Sequential Command Completes and NonSequential Commands Resume	5–81

TABLES

3–1	QIO System Service Parameters	3–15
3–2	SCS message buffer fields	3–45
4–1	MSCP to VMS Error Code mapping	4–11
5–1	Sysgen Parameter MSCP_LOAD settings	5–2
5–2	Sysgen Parameter MSCP_SERVE_ALL settings	5–2
5–3	MSCP Server Data Structures	5–5
5–4	DEC Standard 144 Disk Device Codes	5–17
5–5	Default Load Capacity	5–31
5–6	Routines Invoked by the SEND_DATA Macro	5–47
5–7	Routines Invoked by the REQUEST_DATA Macro	5–49
5–8	Supported Immediate Class Commands	5–54
5–9	NonSequential NonBuffered Commands	5–62
5–10	Supported Sequential Class Commands	5–67
5–11	VMS Error Status to MSPC Status Translation	5–72
5–12	Expected Opcodes for Immediate Commands	5–84
5–13	Expected Opcodes for Sequential Commands	5–85
5–14	Expected Opcodes for NonSequential NonBuffered Commands	5–85
5–15	Acceptable Request States for Connection Failures	5–85
A–1	Data Type Definitions	A–8
C–1	VMS Routine and Module Cross Reference	C–1
C–2	VMS Routine and Module Cross Reference	C–4
C–3	VMS Routine and Module Cross Reference	C–7
C–4	VMS Routine and Module Cross Reference	C–11
C–5	VMS Routine and Module Cross Reference	C–13
C–6	VMS Routine and Module Cross Reference	C–16
C–7	VMS Routine and Module Cross Reference	C–19
C–8	VMS Routine and Module Cross Reference	C–22
C–9	VMS Routine and Module Cross Reference	C–25

Preface

COURSE DESCRIPTION

Overview

This course covers the internals of performing a $QIO to a disk on an "MSCP speaking" controller (e.g. HSCs, ISEs, KDBs, and KDMs), and to a disk which is being served to other VAXes by means of the VMS-based MSCP server. The main focus is on DUDRIVER and the VMS-based MSCP server.

It begins with a brief introduction and overview of SCA and SCS concepts which support the disk class driver communicating with the MSCP disk server in an HSC, ISE, local DSA controller, or remote VAX.

A survey is made of the VMS I/O database related to DSA disks and controllers, and how that database is configured. Then the flow of a typical $QIO for both a read and a write is presented in detail. Included in this flow are the major differences that arise from VMS's point of view due to the different types of DSA controllers.

Next, disk class driver error recovery topics such as mount verification, disk failover, controller reset, the handling of MSCP error status codes, and loss of the SCS connection with the server are studied.

The course then delves into the internals of the VMS-based MSCP server in terms of both functionality and error handling.

Course Format

Lecture/Lab.

Course Length

Five days.

Prerequisites

Formal Training

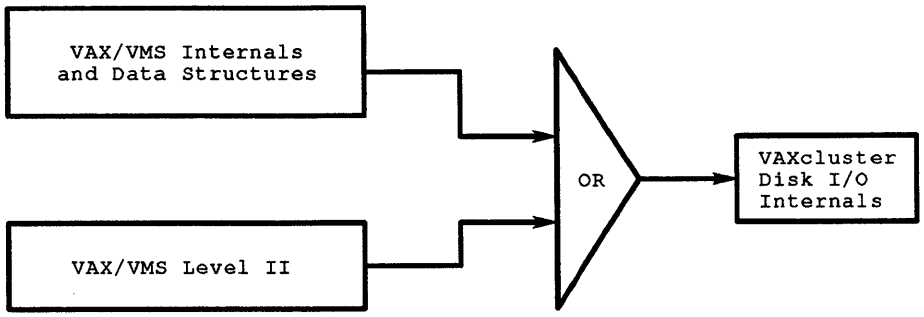
For Software Specialists or Field Service Engineers, at least one of the following is required:

1. VAX/VMS Internals and Data Structures.
2. VAX/VMS Level II.

Experience

At least one year supporting clusterable VAXes.

Curriculum Map



CXN-0000-01

Course Goals

The overall objective of this course is to provide an in depth knowledge of how VMS handles I/O requests and error recovery for disks on "MSCP speaking" controllers. The information presented here should be particularly useful to senior level Field Service and Software Services employees involved in high level technical support, troubleshooting, and VMS crash analysis.

The three major topics to be covered are as follows:

- $QIO flow for both non-shadowed DSA disks and shadow set virtual units.
- Error recovery for both non-shadowed DSA disks and shadow sets.
- VMS-based MSCP server operations.

The emphasis of this course is on the VMS disk class driver and the VMS-based MSCP server. An introduction to SCA, and selected information about DSA controllers and MSCP are provided, but only to the extent that they support an understanding of DUDRIVER, DSDRIVER, SHDRIVER and the MSCP server. This is not a course on the internals of SCA, DSA controllers, or MSCP; there are other courses that cover those topics.

Topic Outline

1. SCA and SCS Concepts Which Support DUDRIVER/DSDRIVER Operations
    1. Datagrams, Messages, and Block Data Transfers
    2. Ports, Virtual Circuits, and Connections
    3. Overview of SCA Architecture
    4. Selected Topics from VMS Implementation of SCA
2. Disk Class Driver Database
    a. Major Data Structures and Their Use
    b. DUDRIVER Database Initialization
3. $QIO Flow for Read and Write Operations
    a. Overview of I/O Channel Assignment and Related Data Structures
    b. Overview of Opening a File and Related Data Structures
    c. Flow of a $QIO for Non-shadowed DSA Disk
4. Error Recovery for Non-shadowed DSA Disks
    a. Disk Class Driver Timeout Mechanism
    b. MSCP End Messages with Error Status Codes
    c. Synchronizing/Resynchronizing with a DSA Controller
    d. Mount Verification
5. VMS-based MSCP Server
    a. MSCP Server's Database
    b. Handling of Commands from Disk Class Drivers in other Hosts
    c. Error Handling

Acknowledgments

The author wishes to express his appreciation to those who have either reviewed the drafts of the material in this book, or contributed with their suggestions. In particular, special thanks goes to Bruce Kelsey of VMS CSSE. Bruce spent an enormous amount of time personally reviewing this material, as well as securing the services of others in this endeavor. Special thanks also goes to Randy Elmer of HSC CSSE for serving as a technical resource in resolving issues related to DSA controllers. Finally, special thanks to Tom Gonzales and Don Smith of the Colorado Springs Training Department for their review of selected chapters.

Chapter 1

SCA and SCS Concepts

1.1 Introduction

Systems Communications Architecture (SCA) was originally designed to serve as an I/O architecture for systems and controllers. The modularity and transportability of its implementation on the VMS operating system has made it the mechanism of choice for inter-system communications for other non-I/O entities as well.

SCA has been optimized for high performance through the creation of specialized communications services and by imposing stringent constraints on topologies. SCA provides the framework for communication among drivers and servers that manipulate devices conforming to the *Digital Storage Architecture* (DSA) standards.

Systems Communications Architecture defines the following:

- The functional layers into which SCA is organized, and their role in providing or supporting the communications services
- The topology (types of configurations) supported
- The types of information exchange
- The "logical" concepts that support the information exchange, such as ports, virtual circuits and connections
- The systems communications services provided
- The interface between each pair of layers.

Systems Communication Services (SCS) is the VMS operating system's implementation of SCA and provides the services to allow the communication of entities within a VAXcluster environment.

1.2 Mass Storage Communications

I/O requests are generated as the result of a need by a host system to communicate with a mass storage device. These I/O requests are passed to a device driver that handles a given "class" of devices. The *class driver* formats the request into *Mass Storage Control Protocol* (MSCP) packets. The class driver utilizes SCS to pass the request to a software layer called the *MSCP Server* in the device's controller where it is resolved.

Class drivers are responsible for the following types of activities:

- Initializing and maintaining the operating system's database for those devices
- Converting I/O requests into commands to be sent to controllers for those devices
- Passing such commands to the software component within the operating system which actually transmits them to the controllers
- Handling responses received from the controller for such commands
- Activating the host operating system's mechanism for passing completion status and other related information to the initiator of an I/O request
- Handling errors related to its class of devices and their controllers on behalf of the host operating system.

MSCP Servers are responsible for the following types of activities:

- Initializing and maintaining various components of the controller's database related to the devices which it is serving.
- Transferring data between a host and a device on the controller.
- Returning to a host current execution or completion status of a command received from that host.
- Maintaining and/or altering the status of a device.
- Reporting controller and device status information to a host.

The VMS class driver for handling disk devices ported to DSA controllers is called DUDRIVER. Certain software extensions to the disk class driver are required to support the controller based shadowing (shadowing phase I) product. When these extensions are required, DSDRIVER is used as the disk class driver in lieu of DUDRIVER.

Several DSA controllers are available. Some examples of controllers which provide the server function are indicated in the following table:

Controller	Application
HSC	CI based Hierarchical Storage Controller
ISE	DSSI based Integrated Storage Element
UDA	UNIBUS based local controller
KDA	QBUS based local controller
KDB	BI based local controller
KDM	XMI based local controller

A VMS host system is capable of emulating a DSA controller through software and can provide the MSCP server function on behalf of its local disks. This action is referred to as MSCP serving and is not limited to DSA compliant devices. The VMS emulation of the server is transparent to the disk class driver and provides access to local disks to all VAXcluster members.

The class driver for the DSA compliant tape devices is TUDRIVER. As with the disk class driver, the tape class driver is responsible for formatting an I/O request into the Mass Storage Control Protocol. To distinguish the tape protocol from the disk protocol, it is referred to as Tape MSCP or simply TMSCP.

Once the TMSCP packet has been created, the tape class driver utilizes SCS to send the packet to a TMSCP server in a DSA controller.

VMS provides for the software emulation of a DSA tape controller in VMS V5.5. Prior to that release, VMS was incapable of providing access to local tape devices to VAXcluster members other than the local host. This access is provided through a TMSCP server SYSAP. As with the disk server, it is transparent to the tape class driver.

1.3 Inter-System Communications

VAXcluster members function in a cooperative and coordinated manner. This requires the exchange and sharing of information between all members.

The *VMS Lock Manager* software provides the tool that allows cooperating processes to synchronize their access to shared resources. Within the context of a VAXcluster, there is a distributed component of the Lock manager which provides this synchronization on a cluster-wide basis. Lock management information is exchanged among the hosts in a VAXcluster within the framework of SCA.

The *VMS Connection Manager* provides the tool that coordinates and controls the membership in a VAXcluster (the *connectivity* of the cluster). The connection manager software on all *active nodes* (VMS nodes) collectively maintain the connectivity of the cluster. The connection managers ensure that all active nodes in the VAXcluster can communicate and consequently coordinate their activities.

When a VAXcluster is first formed, the Connection Manager on one of the hosts assumes the role of "coordinator". The coordinator steps all remaining Connection Managers through the formation of the VAXcluster. When a host joins or leaves a VAXcluster, one of the Connection Managers will act as coordinator in transitioning the other Connection Managers through the change. Connection Manager information is exchanged among the hosts in a VAXcluster within the framework of SCA.

1.4 Information Exchange

Systems Communication Architecture defines three forms of information exchange.

1.4.1 Datagrams

Datagrams are units of information exchange whose delivery is on a "best effort" basis. There is a "high probability" that datagrams will arrive at their destination, but there is no guarantee that they will. Furthermore, there is no guarantee that a sequence of datagrams will be delivered in the same order that they were sent. A user of the datagram service (e.g. DECnet if it is run on the CI) typically performs its own message loss detection and recovery.

1.4.2 Messages

Messages are units of information exchange whose delivery is guaranteed without loss or duplication. Furthermore, the SCA message service is said to be "sequenced". This guarantees that a series of messages all sent with the same priority will be delivered in the same order that they were sent in. The disk class driver uses messages for issuing commands to an MSCP disk server. An MSCP disk server uses messages to send completion status for commands to the disk class driver.

1.4.3 Block Data Transfers

Block data transfers are the direct transmission of data between a named local buffer and a named remote buffer. The block data is guaranteed to arrive completely, or an error condition is indicated to the sender. Typical uses of block data transfers are DSA disk read and write operations.

1.5 Communication Mechanisms

Definitions of three fundamental concepts are essential to the understanding of Systems Communication Architecture: the *port*, the *virtual circuit*, and the *connection*. These terms have specific meaning in the context of SCA and may differ from traditional translations.

1.5.1 SCA Ports

1.5.1.1 Definition

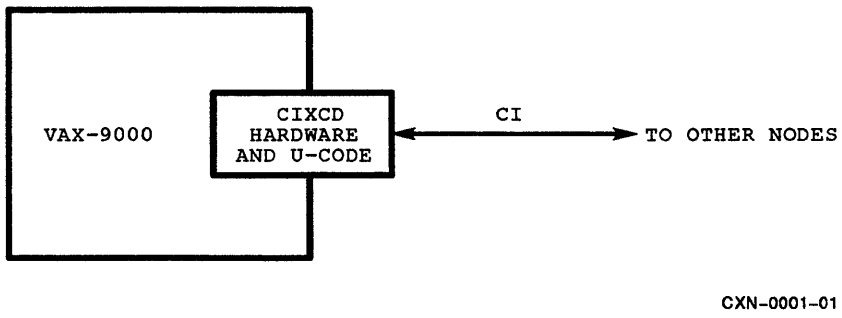
On a given node, an SCA port is the interface between that node and the interconnect providing a physical communication path to the other nodes/servers.

The following table lists some examples of SCA ports:

Interface	Application
CIXCD	XMI bus to CI, VAX 9000 etc.
CI780	SBI bus to CI, VAX 780, 8600 etc.
CIBCA	BI bus to CI, VAX 8350 etc.
SHAC	4000 Cpu module to DSSI, VAX 4000
KDM70	XMI bus to local controller, VAX 9000 etc.
DEMFA	XMI bus to FDDI, VAX 9000 etc.

Figure 1–1 depicts the relationship between the node, the port and the physical interconnect for a typical CI model.

Figure 1–1: CI Node, Port and Physical Interconnect relationship



1.5.1.2 Port Drivers

The VMS operating system provides a software interface to the port called a *Port Driver*. The port driver is responsible for controlling the port as well as exchanging commands and information with the port.

The *Computer Interconnect* (CI) and some *Digital Storage Systems Interconnect* (DSSI) based SCA ports are completely implemented through a combination of hardware and microcode. The port driver for these ports, (*PADRIVER*), is capable of directly manipulating the port hardware and passing information directly to the host memory.

For some implementations of the DSSI port and for the *Network Interconnect* (NI), an additional layer of software is required to interact with the physical port.

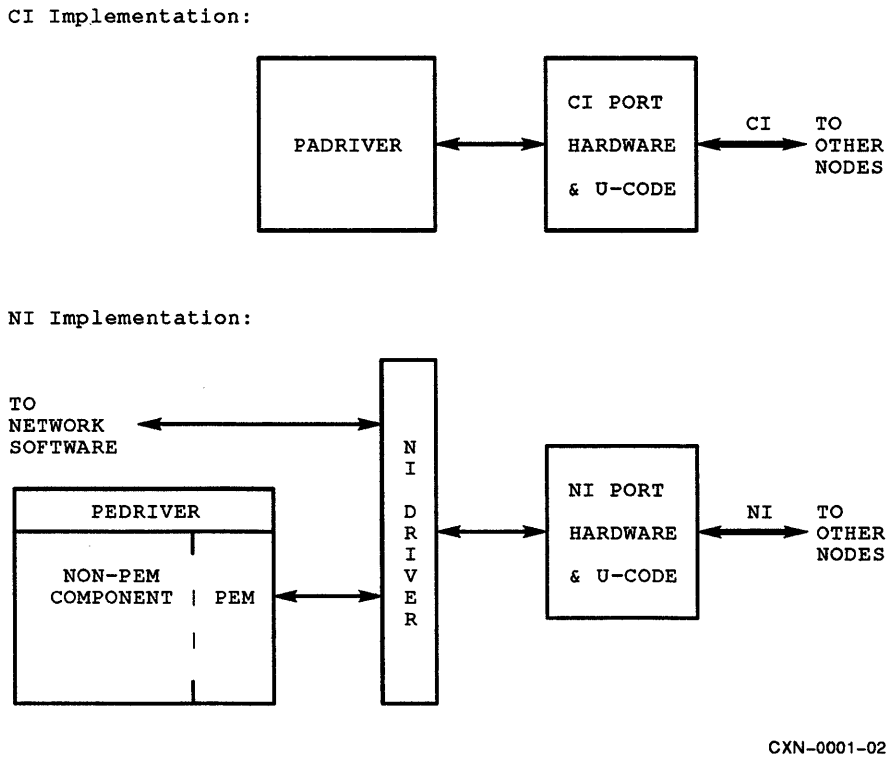
For the DSSI, this additional layer is required for transferring information from the port's local memory to the host's memory. For the NI, the additional software layer is referred to as the *Port Emulator* (PEM) and is used to communicate with a network interconnect device driver.

The port driver for the NI, (*PEDRIVER*), implements this additional PEM software layer. The NI port driver through its Port emulator actually communicates with the physical interconnect through an ethernet (NI) driver. It is the NI driver that is capable of directly manipulating the physical interconnect.

The port driver for some DSSI implementations, (*PIDRIVER*), includes an additional software layer to transfer information between the port and the host's memory. The transferrence requires the host's assistance since the port is incapable of performing the transfer through DMA as does the PADRIVER.

Figure 1–2 depicts the relationship between the port driver and the interconnect for a typical CI implementation and contrasts it with a typical NI implementation:

Figure 1–2: Port Driver, Port and Physical Interconnect configuration for both the CI and NI model



1.5.1.3 Local Controllers

Local DSA controllers are also handled in a similar manner. Local DSA controllers provide both the port functions and the MSCP server functions combined in one controller. The port driver for local devices, (*PUDRIVER*), performs the same type of manipulation of the hardware controller as its CI, DSSI and NI counterparts.

1.5.1.4 Port Descriptors

Corresponding to each port on a node, VMS builds a data structure known as a *Port Descriptor Table* (PDT). Each PDT contains the following types of information:

- Identification of the type of port and characteristics
- The addresses of various queues associated with the port

- The addresses of port specific routines to perform the following types of operations:
  - Buffer allocations for building commands
  - Buffer deallocations
  - Accepting of connections
  - Sending data
- The UCB address for the port

1.5.2 SCA Virtual Circuits

1.5.2.1 Definition

A *Virtual Circuit* is a communication path between two nodes over a physical interconnect. To establish a virtual circuit, the nodes must successfully enter into a dialogue and complete the following three tasks:

- Each node's port identifies itself to the other node
- Each node identifies itself to the other node
- The integrity of the physical path between the nodes is verified.

The virtual circuit will exist as long as the integrity of the communication path remains intact. The path is periodically verified by exercising it in the absence of actual communications traffic.

1.5.2.2 Virtual Circuit Data Structures

VMS utilizes two data structures to maintain the virtual circuits to other active nodes and to *Passive Nodes* (remote DSA controllers). A *System Block* (SB) exists for each node with which the node has communications and a *Path Block* (PB) exists for each port over which the communication may occur. These data structures are maintained by both active nodes as well as passive nodes. Local DSA controllers do not require such structures since their communication is restricted to a local host.

1.5.2.3 System Blocks

The System Blocks are used to describe each of the nodes that are accessible over a physical interconnect. A system block also exists for the local system as well as for each local DSA controller. The following types of information are found in each system block:

- The node's hardware type (e.g. 8800, 9000, HSC90, ...)
- The node's operating system and version (e.g. VMS V5.5-2, HSC V650, ...)
- The node's name

1.5.2.4 Path Blocks

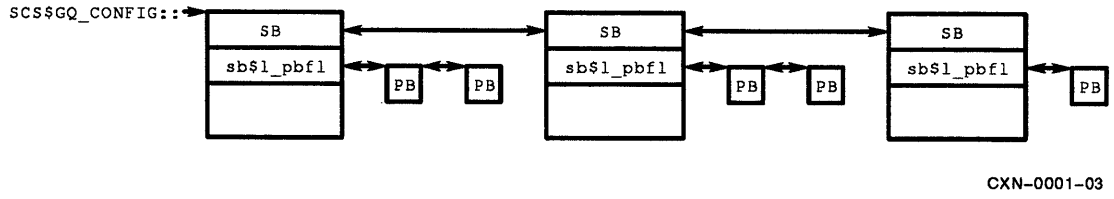
The Path Blocks are used to describe each virtual circuit between the current node and the other nodes. A path block exists for each local system block as well, but contains minimal information since the need for virtual circuit formation does not exist with local devices. The following types of information are found in each path block:

- Indication of the state of the virtual circuit (e.g. open, closed, etc.)
- The address of the port descriptor table
- The local port name
- The remote port type

1.5.2.5 Virtual Circuit Data Organization

Figure 1–3 depicts how the system blocks are linked together off of the system location *SCS$GQ_CONFIG*. The corresponding path blocks are linked from each system block describing the possible paths to the associated node.

Figure 1–3: System Block and Path Block linkage



1.5.3 SCA Connections

1.5.3.1 Definition

To exchange information between entities on two nodes, a logical communication path must exist between the entities involved in the communication. This logical communication path is known as a *Connection*.

Whereas the virtual circuit represents communication between ports, the connection represents communication between entities. Connections utilize virtual circuits as their communications path. An example of a connection would be a disk class driver communicating with an MSCP server. This communication would be performed over a virtual circuit between the two ports involved.

1.5.3.2 Connection Descriptors

As connections are formed, *Connection Descriptor Table* (CDT) entries are allocated on each node and an associated entry is made in the *Connection Descriptor List* (CDL) to contain a pointer to this CDT structure. The offset into this list is the *Connection Identifier* (CONID). Since this operation occurs on each node for a single connection, two connection identifiers result (the local and the remote identifiers). Only the low order 16 bits of these longword connection identifier values is used for the index into the list.

Each CDT contains the following types of information:

- The state of the connection (e.g. open, closed, ...)
- The addresses of the ASCII text strings which provide the names of the two "entities" which are exchanging information by means of the connection.
- The addresses of the routines to which messages and datagrams are to be passed when they are received from the "entity" at the other end of the connection.
- The address of the PB describing the virtual circuit supporting the connection.
- The *Local Connection Identifier* (LCONID) which identifies the corresponding CDT on the local node
- The *Remote Connection Identifier* (RCONID) which identifies the corresponding CDT on the remote node.

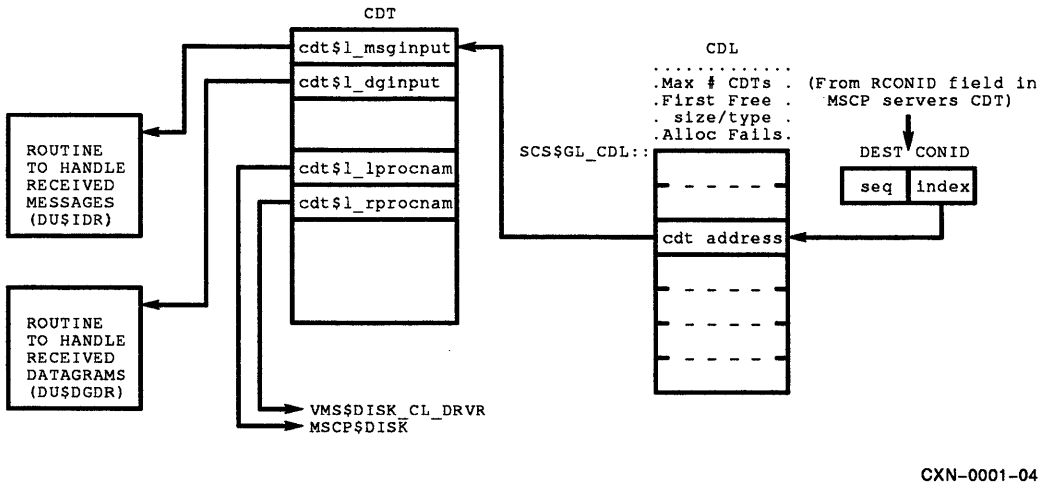
1.5.3.3 Connection Structures

When a message is to be sent from a remote node across a connection, the RCONID value is copied from the remote node's CDT into a destination CONID field within the message. The sending node also copies its local connection id into the message so that the receiving node knows where the message is from.

When this request is received on the local node, the low order 16 bits from the destination CONID are used to locate the associated CDT. Within the CDT will be the address of the routine which is to be executed based on the type of message that was received. Between the node identifiers (SCS system ids) and the connection identifiers, a specific connection can be uniquely identified within a VAXcluster. The CDT and CDL structures are the VMS implementation of the SCA concept of connection blocks.

Figure 1–4 illustrates an example of an MSCP server sending the disk class driver a message:

Figure 1–4: MSCP Server to Class Driver Message Flow

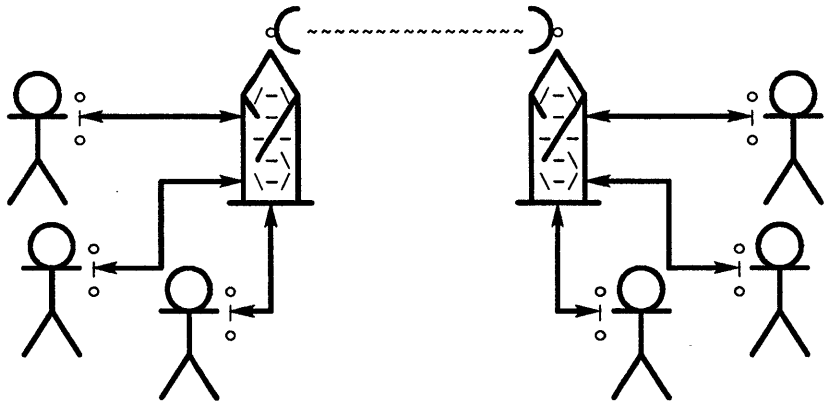


1.5.4 Communications Mechanisms Example

The telephone system provides a good analogy to emphasize the distinction between the concept of a virtual circuit and a connection. Consider the situation wherein three people in one city wish to have phone conversations with three people in another city. Next to each city is a microwave tower. The microwave beam between the towers represents a virtual circuit. The cities represent two nodes, and the microwave towers represent SCA ports. The people then represent the "entities" within the nodes, and their phone conversations represent connections.

Figure 1–5 illustrates the SCA concepts of Virtual Circuits and Connections as applied to the telephone system.

Figure 1–5: Telephone System Analogy to Systems Communications Architecture



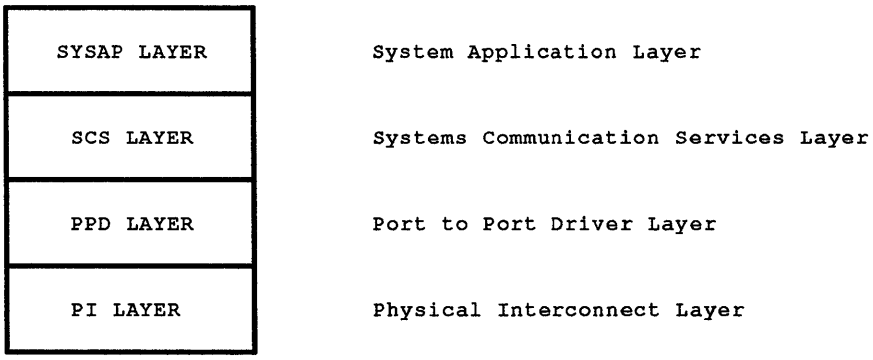
CXN-0001-05

The communication between entities is not restricted to a single *protocol* (format) , much as telephone conversations are not restricted to a given language.

1.6 Architectural Layers of SCA

The traditional model for SCA is organized into four major levels as shown in Figure 1–6.

Figure 1–6: The Architectural Layers of SCA



CXN-0001-06

When extended to support the NI, the SCA model was modified such that the *Port to Port Driver* (PPD) and *Physical Interconnect* (PI) layers were replaced by seven functionally equivalent layers. For the purposes of this discussion, the traditional model of SCA will be used.

1.6.1 SYSAP Layer

1.6.1.1 Definition

The "entities" in a host or controller which utilize the communication facilities defined by SCA to exchange information with their counterparts in other hosts and controllers are known as *System Applications* (SYSAPs).

DUDRIVER and the MSCP disk server are examples of SYSAPs. TUDRIVER and the MSCP tape server are another pair of SYSAPs. The VMS Connection Manager, the distributed portion of the VMS Lock Manager, and a few other VAXcluster specific software components are combined into one SYSAP called *SYS$CLUSTER*. This SYSAP was formerly known as CLUSTRLOA.

SYSAPs may be implemented in software. Such is the case for DUDRIVER and TUDRIVER residing on a VAX, and the disk and tape servers residing on an HSC controller. SYSAPs may also be implemented by microcode. Such is the case for the MSCP disk server in the KDM70.

1.6.2 SCS Layer

The SCS (Systems Communications Services) layer defines the actual services necessary to establish, use, and maintain logical communication paths (connections) among SYSAPs. SCS operations are classified as being either *"port independent"* or *"port dependent"*. Within VMS, the port independent operations are implemented in module SYS$SCS (SCSLOA), and the port dependent operations are implemented in certain portions of the port drivers (PADRIVER, PEDRIVER, PIDRIVER and PUDRIVER).

1.6.2.1 Port Independent SCS Services

Some of the port independent SCS services are as follows:

1.6.2.1.1 Connection Management Services

There are five SCS services invoked directly by a SYSAP to govern the creation and existence of a connection.

CONNECT	Used by a SYSAP to request the creation of a connection with another SYSAP.
ACCEPT	A SYSAP which is the target of a connect request uses this SCS service to accept that request.
REJECT	Instead of accepting a connect request, a SYSAP can reject a connect request and optionally supply a "reject reason".

DISCONNECT	Once a connection has been established between two SYSAPs, either SYSAP may terminate communication by using the DISCONNECT service.
LISTEN	Before one SYSAP can "connect" to another, the other must declare its willingness and ability to handle incoming connect requests.

1.6.2.1.2 Directory Services

The *SCS Directory Service* allows a SYSAP to determine if a particular SYSAP exists on a remote node. A name is associated with each SYSAP to facilitate this lookup. The following table lists some examples of SYSAP names:

SYSAP	SYSAP Name
Disk Class Driver	VMS$DISK_CL_DRVR
Tape Class Driver	VMS$TAPE_CL_DRVR
MSCP Disk Server	MSCP$DISK
MSCP Tape Server	MSCP$TAPE
SYS$CLUSTER/CLUSTRLOA	VMS$VAXcluster

When a SYSAP on one node (host or controller) uses the SCS LISTEN service to declare its willingness and ability to handle connect requests, its name is registered into a "list of listening SYSAPs" on that node. When a message containing a connect request is received from another node, the SCS layer scans this list. If the name of the SYSAP specified in the message as being the target of the request is in the list, the request is passed to that SYSAP. If, however, the name is not in the list, then the SCS layer rejects the request.

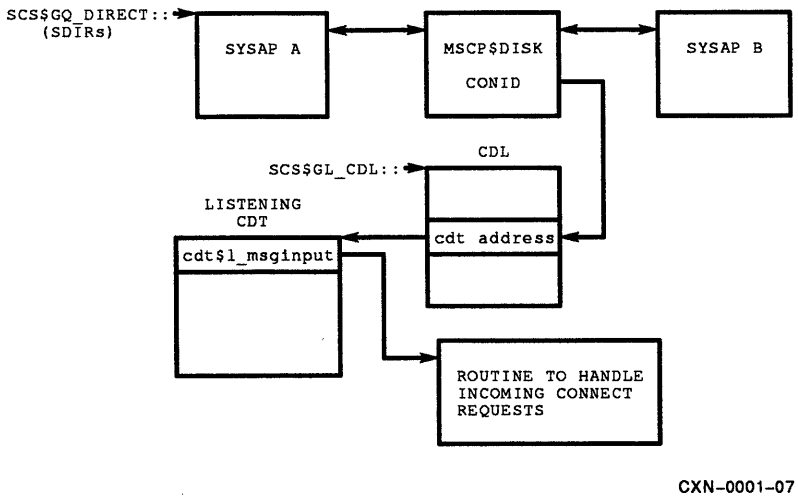
SCA also specifies that each node maintains a special SYSAP to respond to inquiries from other nodes seeking to know if a particular SYSAP name is in its "list" of listening SYSAPs. The name of this special SYSAP is *SCS$DIRECTORY*, and the inquiry is called a *"directory lookup"*.

The implementation of the directory service depends upon the type of host or controller involved. When a SYSAP in VMS invokes the LISTEN service, two data structures are allocated:

- a special *"listening CDT"*
- an *SCS Directory Entry (SDIR).*

The address of the SYSAP's routine for handling incoming connect requests, (supplied by the SYSAP as an argument to the LISTEN service), is stored in the listening CDT. The CONID of the listening CDT is stored in the SDIR along with the name of the SYSAP, and the SDIR is inserted into a queue. Figure 1–7 shows these results for a local VMS MSCP disk server after it has used the SCS listen service.

Figure 1–7: Example of a SYSAP in a Listening State



If the local host receives a connect request for the MSCP disk server, it scans the queue of SDIRs looking for one containing the name MSCP$DISK. From the SDIR it extracts the pointer to the address of the listening CDT. From the listening CDT it obtains the address of the server's routine to which the connect request is to be passed.

1.6.2.1.3 SCS Process Polling Services

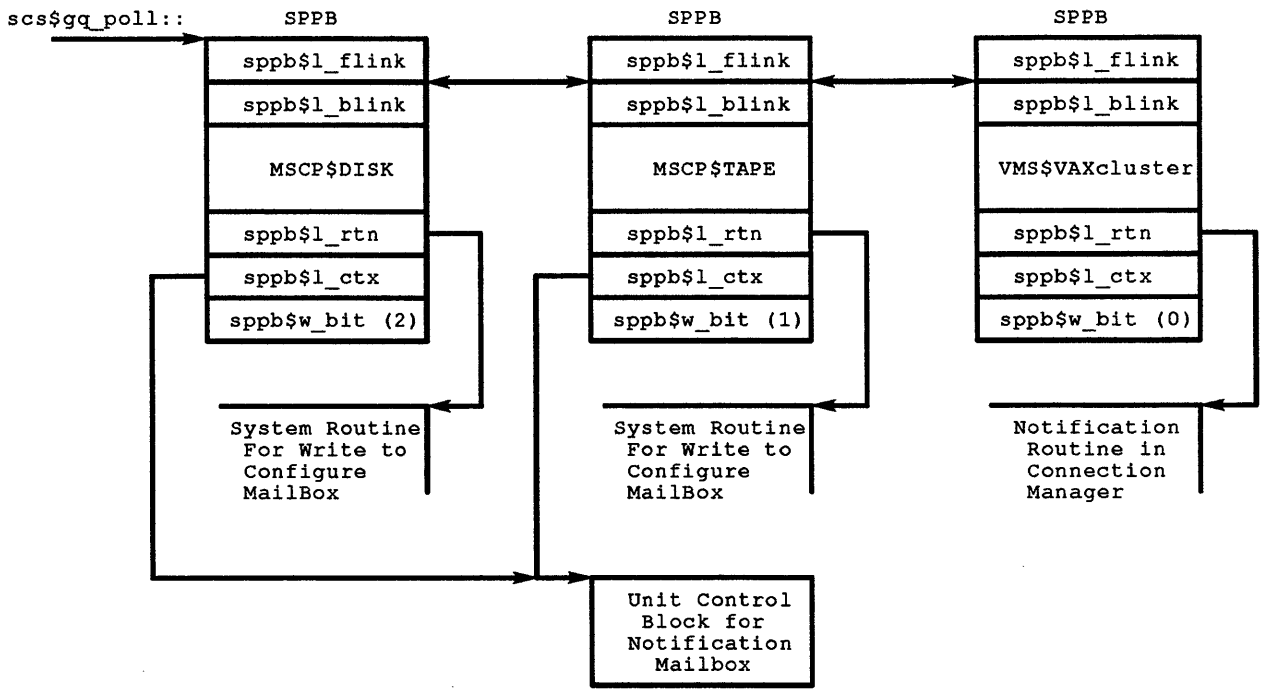
The *SCS Process Poller Service* allows SYSAPs to be notified of the existence of their counterpart SYSAPs that are in a listening state on other nodes. The name of this SYSAP is *SCS$DIR_LOOKUP.*

A SYSAP such as VMS$VAXcluster wishing to be notified of the discovery of a companion SYSAP on another node registers its interest with the local SCS$DIR_LOOKUP service through a call to the *SCS$POLL_PROC* routine. The SCS process poller will periodically connect to the remote node's SCS directory service to determine if the given SYSAP is available in a listening state. If an affirmative response is received, the inquiring SYSAP is notified and a connection request is sent.

The SCS process poller keeps a list of SYSAPs to be polled for in a queue of data structures called *SCS Process Polling Blocks* (SPPBs). Each SPPB is assigned an index number to be used as a bit offset into each *System Block's Enabled Mask* (SB$B_ENBMSK).

When process polling is due, the process poller checks the next *scheduled* System block's mask to determine which SYSAPs are to be polled for on that remote system. For each bit set in the mask, the associated SYSAP name is placed in a *SCS Process Name Block* (SPNB). The name block is then used by the SCS$DIR_LOOKUP service to inquire as to the existence of the given SYSAP on the remote node. Figure 1–8 illustrates the data structures associated with process polling.

Figure 1–8: SCS Process Poll Block Linkage



NOTE

The term "process" is used in this context because the VMS implemenation of SYSAPs is as *fork processes*.

Locating MSCP disk servers is a bit more complex. As will be explained in the next chapter, the *CONFIGURE process* requests SCS$DIR_LOOKUP to poll for disk servers through a call to the *SCS$POLL_MBX* routine. When CONFIGURE is notified by SCS$DIR_LOOKUP that one is found, it builds certain data structures. It then calls the disk class driver's controller initialization routine, passing it these data structures.

1.6.2.1.4 SYSAP Connection Analogy

The act of a remote SYSAP attempting to connect with a local SYSAP is somewhat analogous to a person placing a telephone call to a person in another city. The SCS system id is used to route the request to the appropriate node similar to the telephone caller using an area code to specify the destination city. The SCS SYSAP name is likewise analogous to the specific telephone number within the destination city.

A SYSAP performing the listen service is similar to the act of requesting that a person's telephone number be placed into the local telephone book. The SCS directory service then performs similarly to the directory assistance that is provided by the telephone company. The list of listening SYSAPs can be thought of as a telephone book.

1.6.2.2 Port Dependent SCS Operations

There are a number of SCS operations which by their nature are best handled in the port driver. In fact, some of these are actually port dependent. Here are three SCS operations implemented by port driver routines:

- Allocation and Deallocation of Command and Message Buffers.

  For local DSA controllers, all buffers are pre-allocated during controller initialization. When a message buffer is needed by a SYSAP, an attempt is made to allocate the buffer from a free queue. If the free queue is empty, a "command ring" of buffers containing port commands is searched for a buffer whose "ownership" has been returned by the port to VMS. When a SYSAP releases a received message buffer, the buffer is either placed in a "response ring" for receiving packets from the port, or into the free queue if the "response ring" is full.

  For remote DSA controllers, buffers are dynamically allocated from nonpaged pool for SYSAPs wishing to send messages. When a SYSAP releases received message buffers, they are either inserted into a free queue of buffers for receiving messages from remote nodes, or deallocated to nonpaged pool if that free queue already has a sufficient number of buffers.

- Mapping and Unmapping Block Data Transfers.

  Given a UDA50, traditional UNIBUS mapping registers are used for mapping block data transfers. For a KDA50, QBUS map registers are used. For a KDB50, a software emulation technique using "pseudo-map registers" allows the KDB to be treated similarly to a UNIBUS controller. This is covered in the chapter entitled "$QIO System Service and DUDRIVER".

  When the block data transfer involves a remote controller, then a CI-SCA, a DSSI-SCA or NI-SCA port is involved. The block data transfer is first mapped to system space. A special buffer descriptor is then initialized to indicate where in system space the transfer begins and how large the transfer is. This descriptor is later included in a command passed to the port for processing. A separate section in this chapter provides details on this subject.

- Handling SCS routing information.

  Port driver code is responsible for inserting SCS routing information, such as source and destination CONIDs, into packets being handed to the port for transmission. The port driver also uses that same SCS routing information to deliver received packets to the SYSAPs to which they have been sent.

1.6.3 PPD Layer

The PPD layer provides a number of services, among which are the following:

- Passing packets to and receiving packets from a port.
- Processes commands from the port dependent portion of the SCS layer.
- Initiates the actual transmission of data to remote ports.
- Handles the physical reception of data from remote ports.
- Has responsibility for insuring the integrity of data packets exchanged across the physical communication path between ports.
- Implements the protocol necessary to insure the guarantees associated with messages and block data transfers discussed in an earlier section of this chapter.
- Provides for virtual circuit control.
- Manages the physical communication path between ports.

In general, most PPD activities are implemented primarily by the port. Only the first of those listed above are actually performed by the port driver. The name of this layer is subsequently a bit misleading.

It should be noted that some of these tasks only "appear" to be performed by the "port" for a local DSA controller. This is due to the controller's dual role as both controller and port. With a local DSA controller, there really isn't a remote port.

1.6.4 PI Layer

The PI layer provides the physical communication path managed by the PPD layer. It is implemented by the medium (e.g. CI, DSSI, NI) over which packets are sent and received.

1.7 VMS Implementation of SCA Architectural Layers

The block diagram on the next page illustrates the VMS-specific implementation of SCA as it relates to disk class driver and VMS-based MSCP server operations.

The following items should be kept in mind as the diagram is examined:

- This book is concerned with Systems Communications Architecture only in so far as it supports the activities of the disk class driver and the VMS-based MSCP server. Consequently, SYSAPs such as CNDRIVER (which optionally implements DECnet on the CI) are omitted since they are not germane to the subject at hand. TUDRIVER has been included only because it has been referenced earlier in this chapter.
- The SCS Process Poller and SCS Directory Service are SYSAPs. Hence, architecturally they belong in the SYSAP layer. The VMS implementation actually places them as part of module SYS$SCS (SCSLOA).

  It is important to understand that an architecture defines a unifying functionality and coherent structure to which its different implementations must conform. Implementations may vary on the details of how they provide this functionality and structure.

Consider the VAX 11/785, 8200, 8650, and 9000. They are varying implementations of the same VAX CPU architecture; however, they all implement the same VAX instruction set. This conformity is also true with software.

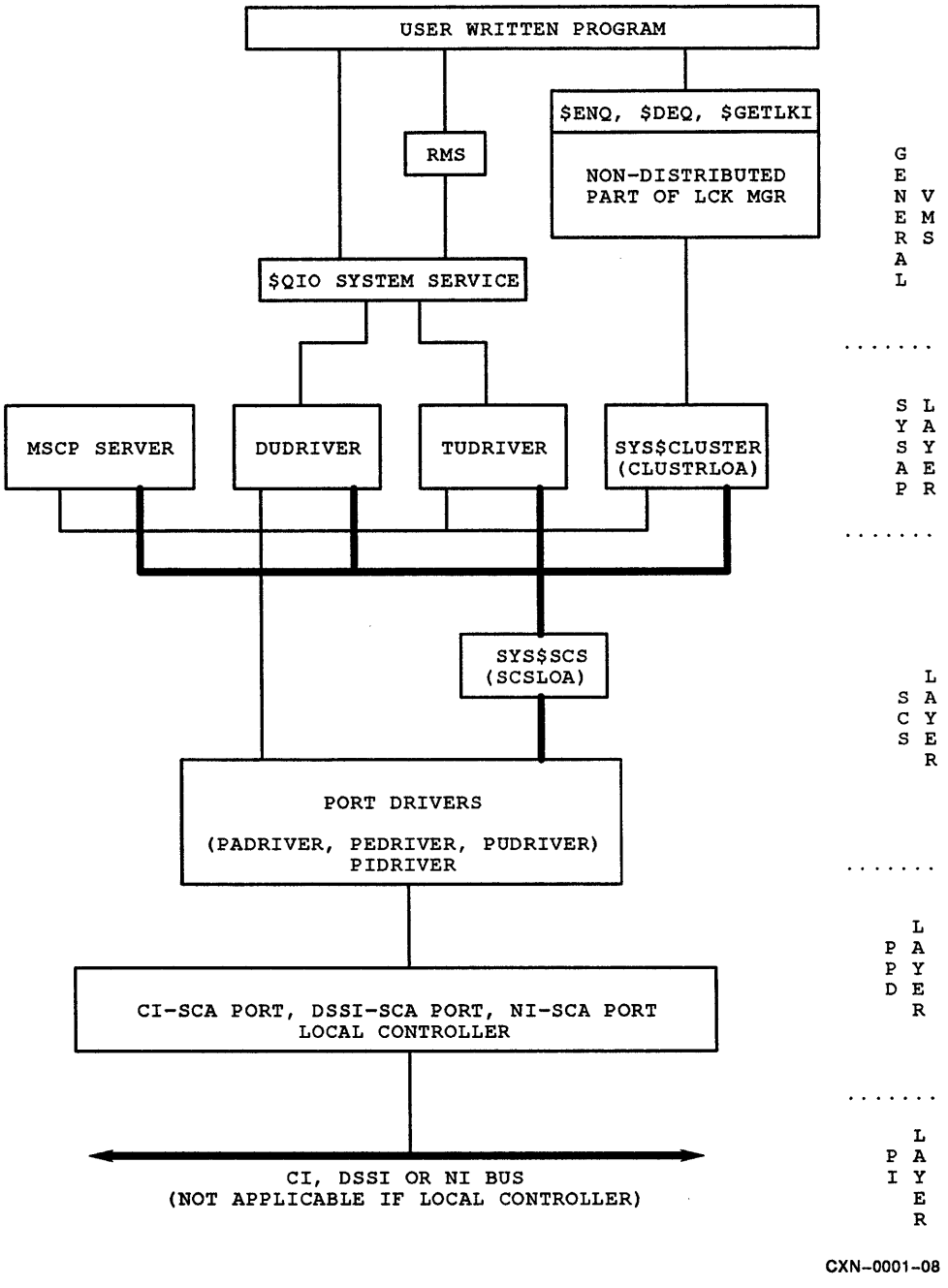
Both a VAX and an HSC implement the SCS Directory Service such that it provides the same architecturally defined functionality; but the details of these implementations vary.

- As was pointed out earlier in this chapter, the VMS Connection Manager, the distributed portion of the VMS Lock Manager, and certain other VAXcluster specific software components are combined into one SYSAP called SYS$CLUSTER (CLUSTRLOA). It should be emphasized that the only interaction the non-distributed portion of the Lock Manager has with SYS$CLUSTER is with its distributed component. There is no interaction between the non-distributed portion of the Lock Manager and the remainder of SYS$CLUSTER.
- VMS supports both shadowed and non-shadowed disks. Only the disk class driver for non-shadowed disks, (DUDRIVER), is shown in the diagram. If controller based volume shadowing is in use, then DSDRIVER replaces DUDRIVER to handle both the shadowed and non-shadowed disks.

The next few sections of this chapter are intended to "tie together" what has been presented thus far about SCA, and do so within the context of how it relates to the disk class driver and the VMS-based MSCP server.

Figure 1–9 provides an overall view of the flow of information for the VMS implementation of SCA:

Figure 1–9: SCA Flow as Implemented on VMS



1.7.1 DUDRIVER CONNECTs to MSCP Disk Server

1.7.1.1 Local Node

When the local SCS Process Poller has discovered a "listening" MSCP disk server on a remote VAX, the *CONFIGURE process* will be notified. The Configure process will in turn call the class driver's controller initialization routine.

The controller initialization routine will call the SCS Connect service (implemented in SYS$SCS (SCSLOA)) to attempt to form a connection with the remote server.

As indicated in the following diagram, SCS will build a "connect request" message and will pass it to the appropriate port driver for transmission.

1.7.1.2 Remote Node

The remote port physically receives the message and passes it to the port driver. The port driver in turn passes it to the remote SYS$SCS (SCSLOA) where the list of listening SYSAPs will be scanned for a corresponding MSCP disk server entry.

If a corresponding entry is found, the connect request is passed to the servers routine for handling connects as found in the listening CDT. The remote SYS$SCS (SCSLOA) will also generate and transmit a connect response to notify the local SYS$SCS (SCSLOA) of the successful reception.

NOTE

If the remote list of listening SYSAPs did not include the server, the response would be a "no such SYSAP" message, and an error would be returned to DUDRIVER by the local SYS$SCS (SCSLOA).

The remote server will check the MSCP protocol being used by the disk class driver against its own and if it is deemed compatible, it will generate an "accept" message.

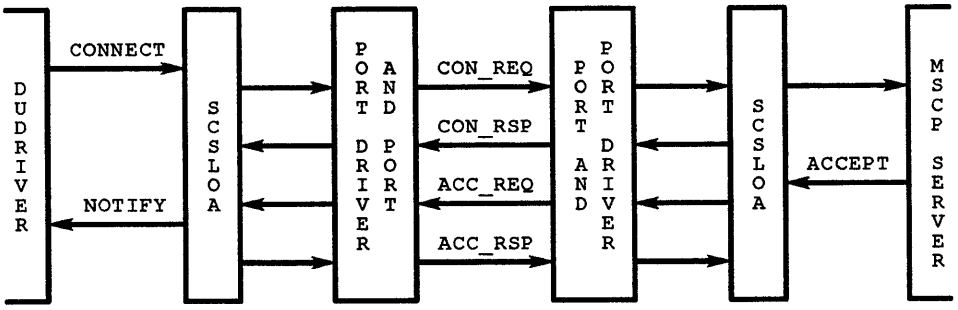
The remote SYS$SCS (SCSLOA) will build an "accept request" message and will pass it to the appropriate port driver for transmission back to the local VAX.

The local port physically receives the message and passes it to the local port driver. The local port driver in turn passes it to the local SYS$SCS (SCSLOA).

The local SYS$SCS (SCSLOA) notifies the class driver that the connect has succeeded and will also generate and transmit an accept response to notify the remote SYS$SCS (SCSLOA) of the successful reception.

Figure 1–10 illustrates a disk class driver forming a connection with a VMS based MSCP server:

Figure 1–10: The Message Flow of a Disk Class Driver Forming a Connection with an MSCP Server



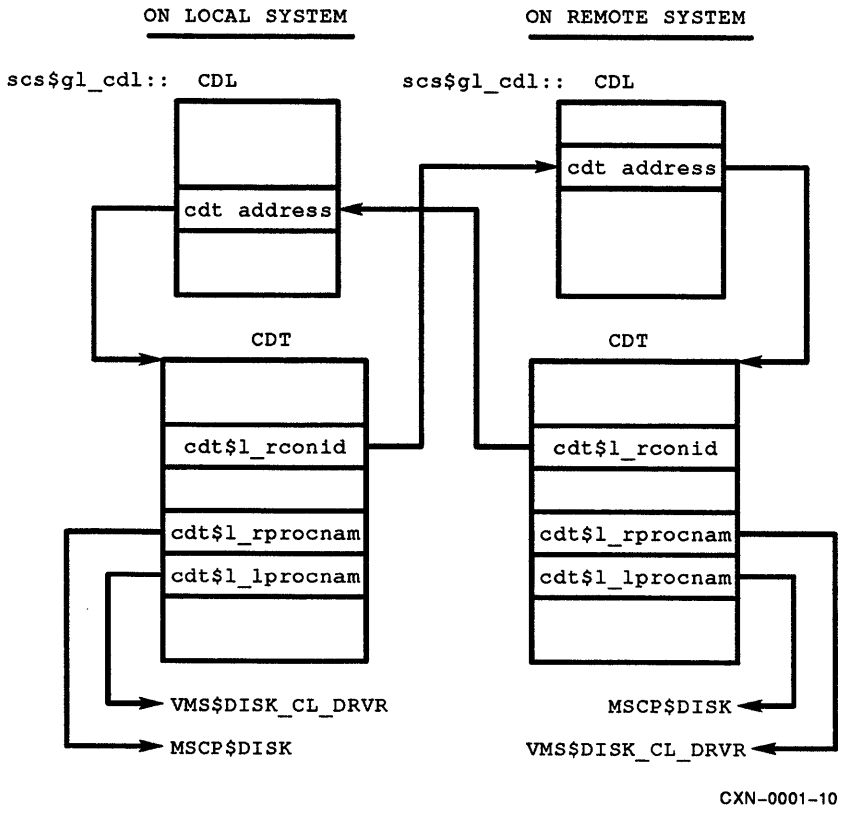
1.7.1.3 Connection Data Structures

A CDT will be built on each VAX that was involved in the connect request to describe the resulting connection. Each VAX will store the following information in its own CDT:

- The CONID identifying the other VAX's CDT. This is called the "remote CONID", and is kept in the RCONID field
- The name of the SYSAP at the other end of the connection on the other VAX. This is called the "remote process name", and is kept in the RPROCNAM field.
- The CONID identifying this VAX's CDT. This is called the "local CONID" and is kept in the LCONID field
- The name of the its own SYSAP involved in the connection. This is called the "local process name", and is kept in the LPROCNAM field

Figure 1–11 illustrates these relationships:

Figure 1–11: Data Structures for a Formed Connection



1.7.2 DUDRIVER Sends MSCP Command to MSCP Disk Server

A user program may request an I/O operation for a disk either by directly using the $QIO system service, or by indirectly using $QIO through RMS. Based on the parameters it is supplied, $QIO builds an *I/O Request Packet* (IRP) describing the operation to be performed. This IRP is then passed to the driver responsible for the type of disk involved. For disks handled by an MSCP server, that driver would be DUDRIVER (or DSDRIVER if the I/O operation is for a controller based shadow set).

1.7.2.1 Buffer Allocation

DUDRIVER allocates a buffer in which to build an MSCP command for the remote server. The routine which does this also copies the RCONID field from the CDT into the buffer's "destination CONID" field; this will facilitate directing the command to the proper SYSAP on the remote node. Using the information contained in the IRP, the routine will complete the build of the MSCP command. Finally, it passes the buffer to the port driver for transmission to the remote node.

1.7.2.2 Identifying the Receiving Sysap and Connection

On the remote node, the port driver extracts the destination CONID field from the buffer containing the received command. The low order 16 bits of the destination CONID are used as an index into the CDL to obtain the address of the CDT used by the remote node to represent the connection. Within the CDT is the address of the MSCP server's message input routine. The message input routine is called, passing it the command received from the class driver on the local VAX.

1.7.3 MSCP Server Sends END Message to DUDRIVER

Mass Storage Control Protocol defines that for each command received by an MSCP server, the server must return an *END message* upon completion of the command. This END message contains completion status and other information, depending on the type of command sent by the disk class driver.

The VMS-based MSCP server doesn't need to allocate a message buffer for this purpose. It merely re-uses the buffer containing the received command, changing selected fields to reflect that what it is sending is in fact the corresponding END message.

When DUDRIVER sent the MSCP command to the server, it not only included a destination CONID, but also a source CONID. The server must also interchange the contents of these fields. It can then pass the buffer containing what is now an END message to its port driver for transmission back to the local VAX.

1.7.3.1 Locating the Connection Associated with an End Message

When the END message is received by the local port, it is passed to the local port driver. The local port driver uses the END message's destination CONID field (which now contains the local CONID) to index into the CDL to fetch the CDT on the local VAX representing the connection with the remote server. It then passes the END message to DUDRIVER's message input routine, the address of which is in the CDT.

1.7.4 Response IDs and Command Reference Numbers

VMS utilizes a *Response Identifier Service* to distinguish which I/O request is associated with each MSCP request. This identifier is part of the VMS implementation of SCS.

1.7.4.1 Class Driver Request Packet

A data structure called the *Class Driver Request Packet* (CDRP) is used by the class driver for each IRP it receives to define a request to be passed to the SCS layer. The IRP will be located at negative offsets from the CDRP information.

1.7.4.2 Request Descriptor Table Entries

The class driver's *STARTIO* routine will allocate an entry called a *Request Descriptor Table Entry* (RDTE) from the *Request Descriptor Table* (RDT) to keep track of each IRP/CDRP pair. The RDTE consists of two longword values.

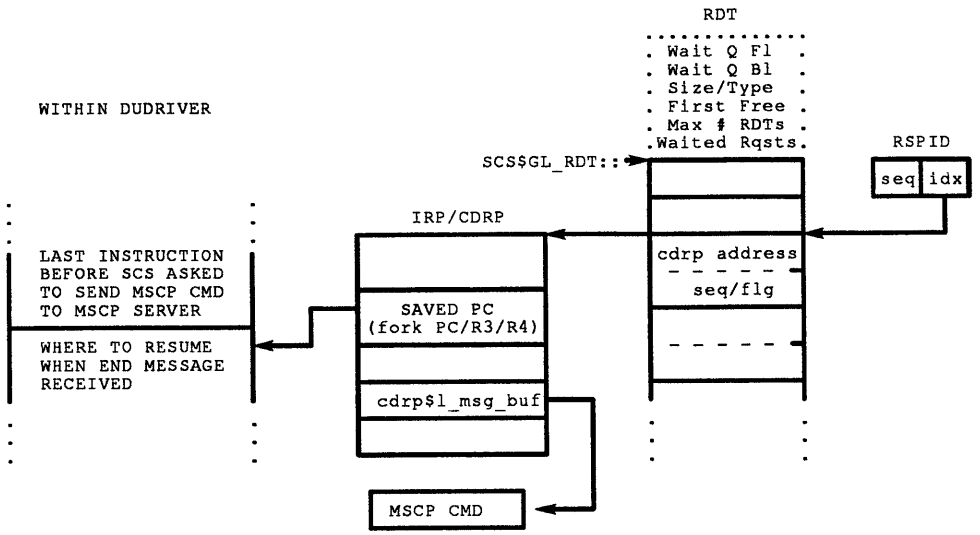
The first longword is initialized with the address of the requests CDRP.

The second longword will hold a value called the *Response Identifier* (RSPID). The low order 16 bits of the RSPID will be an index into the RDT to identify this particular request. This value will be passed in the MSCP command as the *Command Reference Number* field to uniquely identify this request.

The class driver also stores in the CDRP the address of the buffer in which it has built the MSCP command.

When the port driver actually transmits the command, the I/O request is suspended with the PC of where to resume (along with other data) being stored in the CDRP. (Each I/O request is handled within the context of a *fork process*. It is this fork process which is suspended here.) Figure 1–12 illustrates how the correct fork thread is located.

Figure 1–12: Fork Process Thread association through the RSPID



1.7.4.3 MSCP Server End messages

When the MSCP server sends an END message to DUDRIVER corresponding to some MSCP command, it includes in the END message the command reference number supplied in the command. As was pointed out above, this command reference number is actually a RSPID. Once the class driver determines that the message it has received is an END message, it does the following:

- Uses the low order 16 bits of the RSPID to index into the RDT, and then fetches the RDTE associated with the RSPID.
- From this RDTE it obtains the address of the CDRP (and hence also the IRP) associated with the I/O request.
- From the CDRP it obtains the address of where to resume the I/O request, processing the status and other information contained within the END message.

The command reference number also serves another purpose. In situations wherein DUDRIVER must inquire with the MSCP server about the current status of a command, it includes the command reference number (RSPID) in the inquiry. In this way the server knows which command DUDRIVER is concerned with.

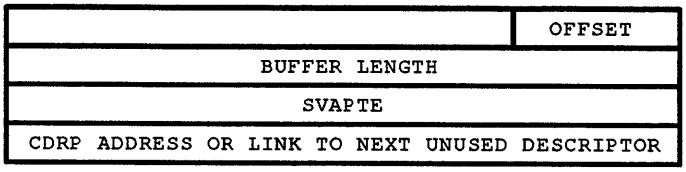
1.7.5 DUDRIVER and Block Data Transfers

Two additional concepts from the VMS implementation of SCS services need to be explained: the *Buffer Descriptor*, and the *buffer handle*. The information presented here is specific to DUDRIVER's dealing with remote DSA controllers (or remote VAXes emulating DSA controllers). Equivalent information for local DSA controllers is presented at the end of the chapter entitled "$QIO System Service and DUDRIVER".

1.7.5.1 Buffer Descriptors

When data is to be transferred to or from a disk, a buffer for holding the data must be locked in host physical memory and a description of the buffer needs to be built. This description, known as a *Buffer Descriptor Table Entry*, consists of four longwords. The buffer descriptors are kept in a *Buffer Descriptor Table* whose listhead is at system location *SCS$GL_BDT*. Fields in the buffer descriptor that are relevant to this discussion are illustrated in the next diagram. These fields are explained in the paragraphs which follow. Figure 1–13 displays the layout of the buffer descriptor.

Figure 1–13: Layout of the Buffer Descriptor



CXN-0001-12

When locked in physical memory, the buffer is also mapped to one or more consecutive pages of system virtual address space. The system virtual address of the first system *page table entry* (PTE) used to do this mapping is stored in the *SVAPTE* (System Virtual Address Page Table Entry) field.

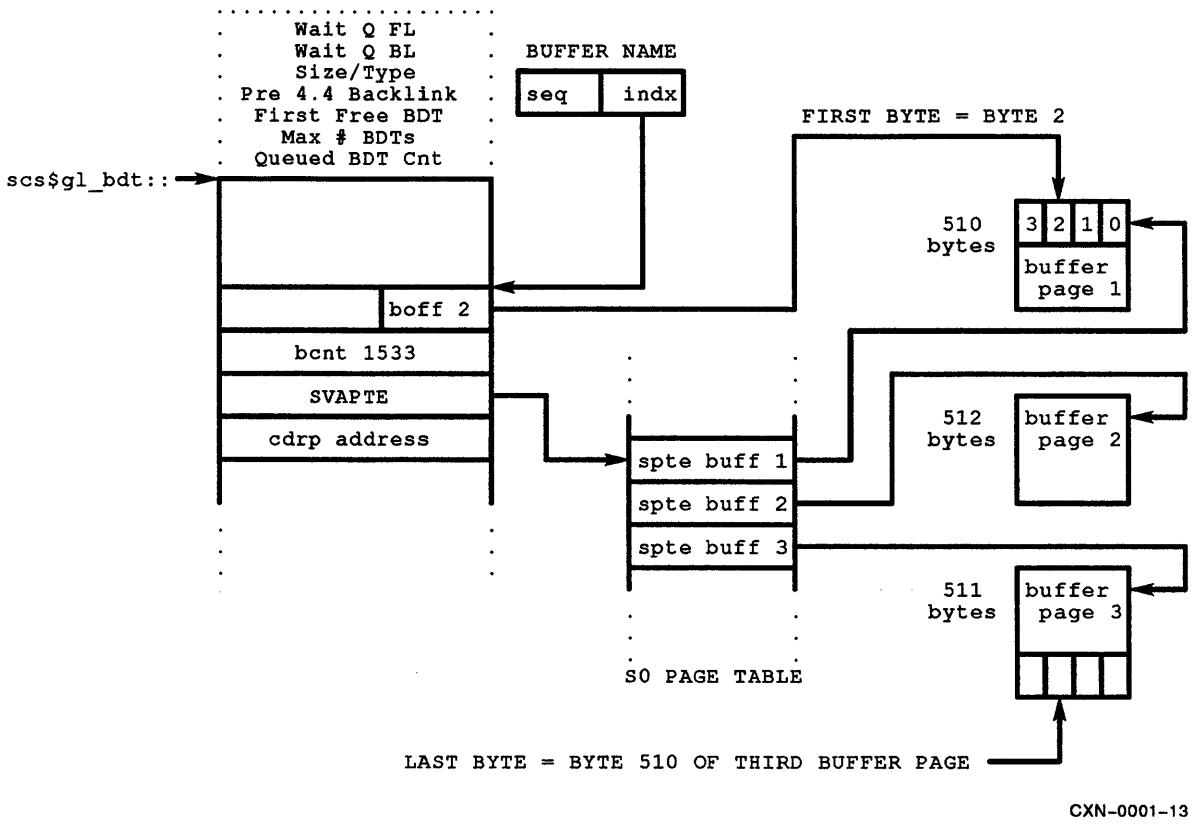
The total length of the buffer in bytes is stored in the *Buffer Length* field.

The buffer does not have to begin on a page boundary. The buffer descriptor provides an *Offset* field to indicate the byte offset into the first page where the buffer actually begins.

The fourth longword serves a dual purpose. For active buffer descriptor entries, it contains the address of the CDRP for the I/O request with which this descriptor is associated. For non-active buffer descriptors, the longword contains a link to the next free buffer descriptor. The listhead of free buffer descriptors is kept at negative offset CIBDT$L_FREEBD from location SCS$GL_BDT.

Figure 1–14 illustrates the use of a buffer descriptor to map a 1533-byte transfer beginning at byte 2 of the of the first buffer page.

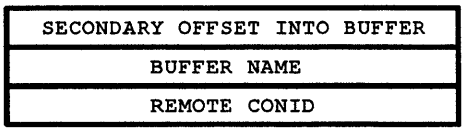
Figure 1–14: Example of a Buffer Descriptor for a Three Page Transfer



1.7.5.2 Buffer Handles

The low order 16 bits of the *Buffer Name* contains an index into the buffer descriptor table to locate a particular buffer descriptor. The buffer name is used as part of still another data structure called a *Buffer Handle*, which consists of three longwords as seen in Figure 1–15:

Figure 1–15: The Buffer Handle



CXN-0001-14

The buffer name and remote CONID have previously been explained. The *Secondary Offset Into A Buffer* may be modified for "third party I/O", but is set to 0 by the class driver at the beginning of an I/O request. The secondary offset field is used for segmented data transfers which will be discussed later.

The buffer handle is inserted into commands that are queued to the local port for the transfer of data to or from a disk on a remote DSA controller (or remote VAX emulating a DSA controller).

The local port functions as a DMA device capable of directly accessing local host memory., It is also capable of performing VAX virtual to physical address translations. Consequently, using the buffer name contained in the buffer handle, the local port can access the buffer descriptor which identifies the transfer address of the buffer in memory.

The local port can extract data directly from or write data directly into the buffer when requested to do so by a remote controller or VAX.

1.7.6 Concept of Flow Control

SCS Flow Control is a mechanism for preventing a SYSAP on one node from sending messages to a SYSAP on another node when that other node is not prepared to accept them. This is accomplished by having the SCS layer on each node "know" how much buffer space is available on the other node for receiving messages.

When two SYSAPs form a connection, each SYSAP requests the SCS layer to allocate to its own port a certain number of buffers for receiving messages from the other SYSAP. As part of the connection formation dialogue, the SCS layer in each node tells its counterpart in the other node how many message receive buffers it has allocated.

1.7.6.1 Credit Scheme

Each SYSAP is said to be extending *"send credits"* to the other. Thus, by requesting the SCS layer to allocate 5 message receive buffers for a connection, the local SYSAP is extending 5 send credits to the SYSAP at the other end of the connection.

Free buffers to receive messages are kept in a common pool, regardless of which SYSAP requested them; but the buffers are managed on a "per connection" basis. This is somewhat like a bank keeping all of its depositors' money in one vault, but assigning an individual non-interest bearing account to each depositor. A person can withdraw from such an account only as much money as he/she has on deposit in that account.

So it is with SYSAPs. At any point in time, a SYSAP should be allowed to receive only as many messages on a connection as it has buffers "on deposit" for that connection in the free buffer pool. Furthermore, SYSAPs should not be allowed to "borrow" receive buffers from one connection to be used for another connection; and they certainly should not be allowed to practice "deficit spending".

NOTE

For CI-SCA ports, DSSI-SCA ports and NI-SCA ports, this pool is a queue called the *Message Free Queue*. For a local DSA controller, a *"Buffer Ring"* serves this purpose. Details of this ring are presented in the chapter entitled "$QIO System Service and DUDRIVER".

The SCS layer in each node has the responsibility for enforcing the rules of flow control. It uses the CDT as a "passbook" to keep track of the number of available buffers "on deposit" for the connection. Specifically, the CDT's send credit field indicates the number of message receive buffers the local SCS layer "believes" are available in the remote node for the connection.

To request a buffer in which to build a message, a SYSAP calls a routine in the SCS layer. This routine checks the send credit field in the CDT for that connection. If no send credits are available, the SCS layer suspends the request. If there is at least one send credit, the field is decremented and the buffer is allocated to the requesting SYSAP.

1.7.6.2 Piggybacking

As the remote node processes received messages, it returns the buffers back to its SCS layer. The remote SCS layer records in its CDT associated with the connection the send credits possessed by the local node for that connection. When the remote SYSAP sends the local SYSAP a message, the remote SCS layer stores in a protocol defined field within the message the actual send credits available to the local SYSAP. The local SCS layer updates its CDT with this information. This technique is sometimes called *"Piggybacking"*.

If most of the message traffic is "one way", then another mechanism must be used to update the local node. As the remote SYSAP releases buffers back to its SCS layer, the local node won't be updated if there is no traffic returning to the local node. Each time the remote SYSAP releases a message buffer, its SCS layer checks to see if the local node's send credit is getting "dangerously low". If so, it will generate a *"special credit"* message to update the local SCS layer about send credits it has for this connection. The definition of "dangerously low" is SYSAP dependent, but it is typically around 2.

NOTE

Each node uses the CDT *"receive credit"* and *"pending credit"* fields to keep track of the other node's send credits. The receive credit field reflects the other node's send credit. Each time a message is received, the receive credit field is decremented. The pending credit field counts the number of buffers returned by the SYSAP to the SCS layer, but which the other node does not yet know about. Any time one node updates another node's send credit, it computes the sum of the receive credit and pending credit fields, and sends this sum as the "actual send credit" to the other node. It also stores this sum in its own CDT receive credit field, and zeros the pending credit field.

1.7.7 MSCP Server in a Controller

1.7.7.1 Local Server Handles SCA Events Essentially the Same

These sections presenting the "VMS Implementation of SCA Architectural Layers" presumed that the disk class driver resides in the local VAX, and that the MSCP disk server with which it communicates resides in a remote VAX. If, instead, the server resides in a remote or local DSA controller, then the events just described would remain essentially unchanged except where explicitly noted.

The controller would have its own equivalent microcode and/or software implementation for the tasks described here as being performed by remote VAX's SYS$SCS (SCSLOA) and port driver. But it would still all look the same from the class driver's point of view since the remote VAX's VMS-based MSCP server is merely emulating a DSA controller.

Chapter 2

DUDRIVER I/O DATABASE

2.1 Introduction

There are four major VMS data structures used by the disk class driver to keep track of "MSCP speaking" controllers, VMS systems which emulate such controllers, and the disks they make available to the nodes on the CI, DSSI and/or NI.

System blocks and class driver data blocks provide information about controllers DUDRIVER deals with. Device data blocks contain information about classes of devices on a controller. Unit control blocks provide the disk class driver with information specific to particular units.

To follow the details of a $QIO operation for a disk handled by DUDRIVER, it is essential to understand what is in each of these structures and how they are linked together. It is also very useful to understand the steps involved in building DUDRIVER's database as "MSCP speaking" controllers are discovered by the local VAX.

2.2 Data Structures

System Blocks contain the identifying hardware and software information about "systems" which is essential in facilitating SCS communication between SYSAPs in those systems. These system blocks are created and maintained by the SCA layers of software beneath the SYSAPs. They do not contain SYSAP specific information, but rather "system level" hardware and software information needed by Systems Communication Services to support communication between any pair of SYSAPs.

Different SYSAPs, such as disk and tape class drivers, must maintain information about a system which is specific to the SYSAP's nature and function, and beyond that which is kept in a system block. The *Class Driver Data Block* (CDDB) serves this purpose for DUDRIVER. It supplements the information contained in a system block with queues and information specific to the handling of MSCP commands issued by DUDRIVER to the system represented by a system block.

A *Device Data Block* (DDB) contains information applicable to a generic class of devices attached to a single controller. For example, DUDRIVER would maintain one DDB for all disks called "DUA" on one controller, another DDB for all disks called "DJA" on that same controller, and a third DDB for all disks called "DUA" on some other controller.

A *Unit Control Block* (UCB) contains information specific to a particular unit within a generic class of units attached to some controller.

For each system/controller with which DUDRIVER is speaking, there will be one SB and one CDDB. For each generic class of disks on that system/controller pair, there will be one DDB. And for each disk within a generic class, there will be one UCB.

There will be additional DDBs and UCBs supporting shadow set virtual units. However, while some references to volume shadowing are made in this chapter, the topic in general is covered in a later chapter.

2.2.1 SB - System Block

VMS maintains a "configuration list" of System Blocks to describe the local host, as well as "remote systems" with which it communicates via Systems Communication Services (SCS). The phrase "remote system" here is not limited just to other VAXes. However, in the event that the remote system is a VAX, here are some of the typical items of information about that system found in its corresponding SB:

- Nodename (assigned when the software is installed on that node).
- Hardware type (e.g. "9000", "8800", "780", ...).
- Hardware version.
- Software type (i.e. "VMS").
- Software version.
- When the system was initialized.
- Queue of path blocks describing available SCS communication paths between the local host and the remote system.

As indicated above, the term "system" is used here in a more general sense than just the traditional notion of a host VAX CPU running the VMS operating system. For example, HSC40, HSC50, HSC60, HSC70 and HSC90 intelligent controllers have SYSAPs which communicate with VAX-based SYSAPs by means of the standard SCS services and associated protocol. An example of this would be the VMS disk class driver, VMS$DISK_CL_DRVR, communicating with the MSCP disk server, MSCP$DISK, in an HSC90. (VMS$DISK_CL_DRVR is implemented by the VMS DUDRIVER code.)

VMS maintains an SB for each HSC with which it communicates. The system block for an HSC is functionally equivalent to one for a VAX; but certain fields reflect obvious differences. For example, the hardware type for an HSC50 would be "HS50", and the software type would be "HSC".

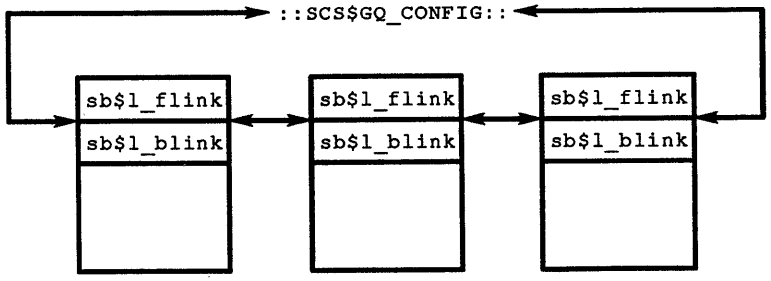
HSCs are merely one category of DSA controllers containing SYSAPs with which the VMS SYSAPs communicate via SCS. Other DSA controllers, such as UDAs, KDAs, KDBs, and KDMs also implement SYSAPs. This implementation is done by means of controller microcode rather than software. VMS SYSAPs communicate with the SYSAPs in local DSA controllers using SCS services in the same manner as they would with SYSAPs in remote VAXes and HSCs. For example, there is a microcoded SYSAP called MSCP$DISK in a local UDA50 to which the local disk class driver issues commands for RA-type disks on that local controller.

By means of SCS services, the local UDA50 "appears" to DUDRIVER as if it were a remote system running an MSCP disk server. Consequently, VMS maintains an SB for each such local DSA controller. Certain fields within this SB are not used (e.g. software and hardware types, software and hardware versions, node name, etc...).

2.2.1.1 Configuration List of System Blocks

As pointed out at the beginning of this section, all the system blocks maintained by VMS are kept in a queue called a "configuration list" as depicted in Figure 2–1. The head of this queue is location *SCS$GQ_CONFIG*.

Figure 2–1: System Block List



A system block representing the local VAX is setup during VMS initialization by module INIT, and begins at global location *SCS$GA_LOCALSB*. Consequently, it is the first SB placed in this queue and is called the permanent local System Block.

System blocks corresponding to remote VAXes and HSCs are allocated and initialized when these remote systems are discovered on the SCS communication medium (CI, NI, ...). How remote systems are discovered depends on the type of communications port used.

2.2.1.2 System Blocks and CI Ports

If a pair of systems are using CI ports to communicate, then each system's CI port driver (PADRIVER in VMS, and CIMGR in an HSC) periodically polls for all other possible ports on the CI. Each VAX and HSC periodically issues a Request ID (REQID) packet for every other possible node on the CI.

When a CI port in an enabled state (or any of the maintenance states) receives a REQID, it responds with a packet which identifies itself to the system which issued the REQID. Virtual circuit formation dialogue occurs between two CI-based nodes as a result of each node receiving a response (IDREC) to its REQID. During this dialogue, each node describes its own CPU and operating system to the other node. As a result of this dialogue, each node allocates

and initializes a system block corresponding to the other. The information in the system block comes from the packets exchanged during this dialogue.

2.2.1.3 System Blocks and NI Ports

A system using an NI port for SCS communication has an NI port driver which interfaces between the SYSAPs and the NI controller software. In VMS, the NI port driver is PEDRIVER, but the NI controller software depends upon which NI controller is used. Two examples would be XEDRIVER for a DEUNA and DELUA, and EXDRIVER for the DEMNA. For purposes of this discussion, the name "NIDRIVER" will be generically applied to the NI controller software.

PEDRIVER consists of an SCS component and a PEM (port emulator) component. When a SYSAP on one system wishes to use SCS to exchange information across the NI with its counterpart SYSAP on another system, it interfaces with the SCS component of PEDRIVER in the same way it would with PADRIVER or PIDRIVER. Thus, it is effectively transparent to the SYSAP whether a CI, DSSI or NI is being used. The PEM component of PEDRIVER has the responsibility of emulating a CI port; it makes the NI controller and associated NIDRIVER appear like a CI port to the SCS component of PEDRIVER.

When a local SYSAP wishes to send a message to a remote SYSAP, it builds the message in a buffer formatted according to its own SYSAP-dependent protocol and then passes the buffer to the SCS component of PEDRIVER. PEDRIVER's SCS component adds standard "CI port style" SCS protocol bytes and inserts the packet into what "appears" to be a standard CI port command queue. The PEM portion of PEDRIVER, emulating a CI port, removes the packet from the command queue and passes it to NIDRIVER in the appropriate manner for transmission on the NI.

When NIDRIVER receives an incoming SCS packet from the NI controller, it passes it to the PEM component of PEDRIVER. There the packet is reformatted to look like it was received from a CI port and inserted into a standard CI *response queue*. The SCS component of PEDRIVER removes the packet from the response queue, strips away the "CI port style" SCS protocol bytes, and passes the remaining buffer to the SYSAP for which it is intended.

The PEM layer of a system's PEDRIVER periodically issues multicast *HELLO* messages for two purposes. One is merely to inform other nodes on the NI that the local node is still "alive and well". The other purpose is to initiate the dialogue and exchange of information necessary to establish communication between the NI port drivers in the two nodes.

Based on this exchange of information, the PEM component of PEDRIVER fabricates a "CI port style" *IDREC* and passes it to the SCS component. From this point on, SCS virtual circuit formation across the NI appears the same as on the CI. The SCS components of the two nodes' PEDRIVERs exchange the CPU and operating system information necessary for each node to build a System Block describing the other node's CPU and operating system using the *Start/Stack/Ack* dialogue.

2.2.1.4 System Blocks and Local DSA Controllers

The disk class driver uses SCS for the exchange of MSCP packets with the disk server in a local DSA controller's microcode. To make this possible, SCS routines must have access to the same data structures that would be present with a normal SCS virtual circuit.

One of the initialization steps performed for each local DSA controller by routine INIT_CTLR in module PUDRIVER is the building of a system block for the controller. However, this is done without the dialogue that is present in virtual circuit formation across the CI, DSSI, or NI. Such dialogue is unnecessary since these steps are taken as part of the normal configuration of local devices performed by VMS as it boots.

2.2.2 DDB - Device Data Block

A Device Data Block (DDB) identifies a generic class of devices and associated controller name (e.g. DUA, DJA, DRB, DIA, ...) attached to a single controller. Some of the items of information found in a DDB are:

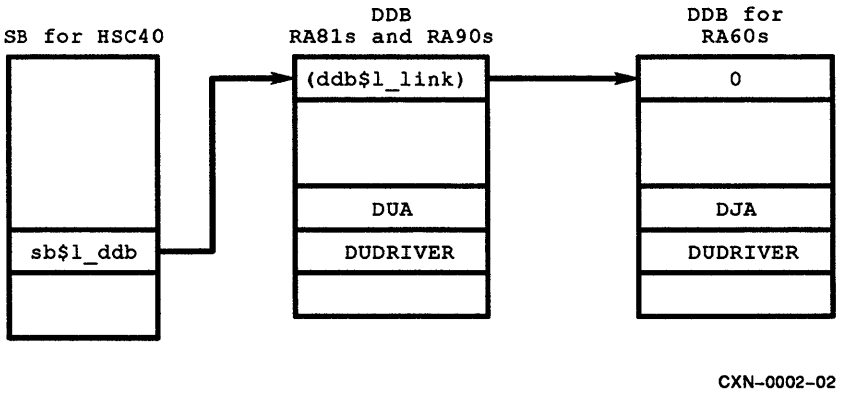
- Generic name of the class of devices represented by the DDB.
- Name of the device driver for the controller.
- Allocation class.

2.2.2.1 DDBs and Remote DSA Controllers

For a remote DSA controller such as an HSC90, the DDBs for that controller's disks are kept in a list which is linked to the system block representing that controller. The head of the list, at offset *SB$L_DDB* in the SB, contains the address of the first DDB in the list. At offset *DDB$L_LINK* of each DDB is the address of the next DDB in the list.

Figure 2–2 illustrates a case of where the DSA controller is an HSC40 having RA81s (DUA), RA90s (also DUA), and RA60s (DJA).

Figure 2–2: DDB Linkage off of the System Block



Each DDB also contains at offset DDB$L_SB the address of the SB to which it is linked.

2.2.2.2 DDBs and Remote MSCP-Served Disks

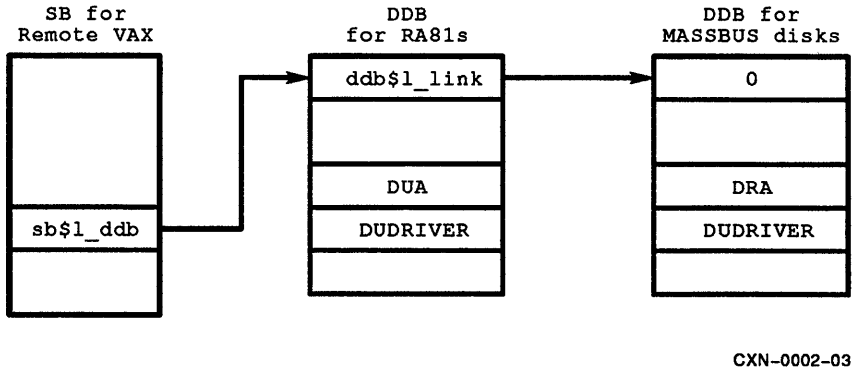
By default, any disk on an HSC is available to any VAX on the CI. However, a VAX may also make one or more of its own disks available as cluster-wide resources as well. That VAX merely has to run the *VMS based MSCP disk server (MSCP.EXE)*, and then explicitly set "served" those of its own disks which it wishes to make available to other VAXes. A VAX may set served only its own disks, and not those attached to some other VAX or HSC. Furthermore, this may be performed across both the CI and the NI.

As an example, assume that a remote VAX is serving a MASSBUS disk to the cluster. A user process on the local VAX wishing to read or write that disk merely issues a $QIO request just as if that disk was being served by an HSC. After certain initial I/O pre-processing, the $QIO system service code passes the request to DUDRIVER. DUDRIVER then builds an MSCP command corresponding to the request and passes that command to the SCS routines for transmission to the MSCP server on the remote VAX.

There is no difference between the format of an MSCP command sent to an HSC and one sent to a remote VAX running the VMS based MSCP server. The remote VAX is "emulating" an DSA controller for such I/O requests.

Suppose that a remote VAX is serving two MASSBUS disks (DRA2 and DRA3) and one RA81 (DUA5) to the cluster. Then the local VAX allocates and links two corresponding DDBs to the SB it maintains to describe the remote VAX. This is illustrated by Figure 2–3.

Figure 2–3: DDB Linkage for a Local Served Disk



Note in the above diagram that the local VAX still uses DUDRIVER when dealing with a remote VAX's MASSBUS disks. If the remote VAX receives an MSCP command for one of its MASSBUS disks, it will convert the command into an equivalent MASSBUS operation and pass it to its own DRDRIVER (the MASSBUS disk driver).

Also note that if the remote VAX had been serving another MASSBUS disk called DRB3 in addition to the above, the local VAX would allocate a third DDB corresponding to "DRB" and link it in with the other two.

2.2.2.3 DDB Chain for Local DSA Disks

If a VAX has a local DSA controller (UDA, KDA, KDB, KDM, ...), it will have an SB corresponding to that controller. However, there will be no DDBs linked to that SB, and the SB$L_DDB field will contain a 0.

DDBs for local DSA disks will be included in the list of DDBs for all local devices. The head of that list is at offset *SB$L_DDB* in the system block that the local host maintains to describe itself.

Location *IOC$GL_DEVLIST* traditionally served as the head of the list of all DDBs for local devices before clustering. It still continues to serve in that role. Therefore, offset SB$L_DDB in the local system block and location IOC$GL_DEVLIST should be expected to contain the same address, namely the address of the first DDB in the chain of DDBs for all local devices. The DDB$L_LINK field of each DDB is a forward pointer to the next DDB in the list, with the last DDB in the list having a 0 in this location.

2.2.2.4 DDB for Boot Device

Location *SYS$AR_BOOTDDB* contains the address of the DDB for the boot device, no matter what type of controller that device is on.

2.2.3 UCB - Unit Control Block

VMS creates and maintains a *Unit Control Block* (UCB) corresponding to each device unit it accesses. The general purpose of a UCB is to specifically identify the unit, describe its characteristics and status, and provide information as to the controller, driver, and current outstanding I/O activity. Here are some of the most frequently referenced items in a UCB:

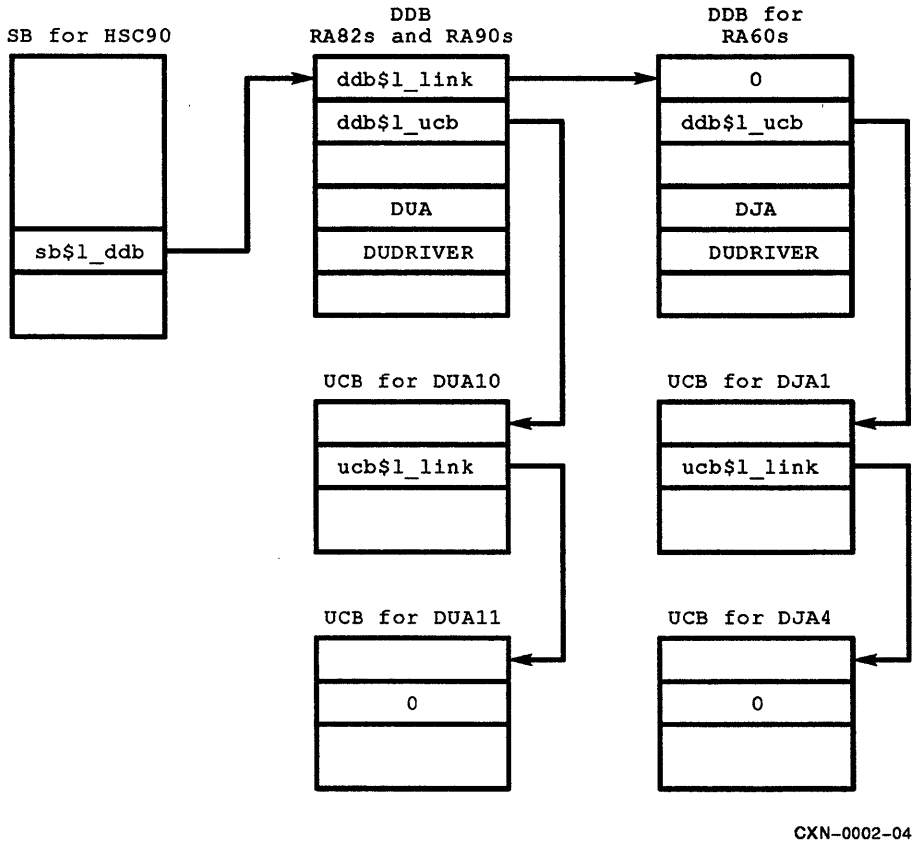
- Media identification (i.e. RA60, RA92, ...).
- Unit number.
- Characteristics flags that indicate such things as whether the device is
  - — Directory structured.
  - — Shareable.
  - — Capable of providing input.
  - — Capable of providing output.
  - — Available cluster-wide.
  - — Dual-pathed.
  - — A member of a shadow set.
  - — MSCP-served.
- Status flags that indicate such things about the device as the following:
  - — Device is online.
  - — Unit has timed out.
  - — Power failed while unit was busy.
  - — Volume on this unit is software valid.
  - — Mount verification is in progress for the volume on this device.
- For traditional non-DSA disks, the address of the IRP currently being processed on this device; and for DSA disks, the address of another data structure (the CDDB) containing the queue of all active IRPs for this and other units on this unit's controller.

2.2.3.1 Linked Lists of UCBs

UCBs are kept in linked lists. The head of each such list is at offset *DDB$L_UCB* in the DDB describing the generic device class and linked to the SB corresponding to the controller to which the unit is attached. Offset *UCB$L_LINK* in each UCB provides the address of the next UCB in the list. A UCB with its LINK field being 0 is at the end of a UCB list.

Figure 2–4 shows an example of an HSC90 with four disks: an RA82 called DUA10, an RA90 called DUA11, and two RA60s called DJA1 and DJA4.

Figure 2–4: DDB Linkage Showing Four Disks



Each UCB also contains at offset *UCB$L_DDB* the address of the DDB to which it is linked.

2.2.3.2 UCBs for DSA and MSCP-Served Disks

The length of a UCB depends on the type of device it describes. All UCBs begin with a common set of fields. Beyond those common fields are various UCB extensions based on the device type.

There are five *UCB extensions* for UCBs representing DSA disks (both remote and local), and also UCBs representing disks on remote VAXes which are serving those disks to the VAXcluster. These extensions are as indicated in the following list:

- Error Log Extension.

Common to all disks, this region contains information useful for logging errors related to the unit represented by the UCB, such as the address of an error message buffer. However, most of the fields in this region are not used for DSA disks.

- Dual Port Extension.

  This UCB is present for all disks, even if they are not dual ported. It indicates if they are dual ported. And if so, it provides secondary path information, such as the address of a secondary DDB linked to the SB representing the other controller.

- Standard Disk Extension.

  Common to all disks, this extension of the UCB contains items such as:

  — The number of times this unit has been placed online since VMS booted.

  — Maximum number of logical blocks on a random access device. (For DSA disks, this is the number of logical blocks available for host data storage.)

  — Maximum transfer byte count.

- MSCP Extension.

  This extension is appended to a disk UCB if either

  — The disk is a DSA-type disk (remote or local), or

  — The disk is on a remote VAX which is serving it to the cluster.

  Some of the items of information provided by the MSCP UCB extension include the following:

  — Address of active/primary class driver data block (CDDB) containing the queue of all active IRPs for the controller currently handling this disk.

  — Address of a secondary CDDB for failover to another controller, if there is a secondary controller ported to this unit.

  — Address of the CDT representing the connection between the disk class driver on the local node and the MSCP disk server in the controller handling this disk (or the MSCP server in the remote VAX which is acting as a "logical controller" by MSCP-serving this unit to the cluster).

  — MSCP unit number.

  — Various MSCP unit flags which indicate such things as whether the disk is formatted for 512 bytes or 576 bytes per sector, if the media is removable, if the unit is write protected, etc.

  — Virtual Unit Pointer to Host Based Shadowing SHAD (discussed in a later chapter)

- "Special" DUDRIVER Extension.

  This is DSA specific information. Much of it relates to the geometry of DSA disks and is used by volume shadowing to insure that proposed members of a shadow set in fact have the "same geometry". Three of the items of information are:

  — Number of LBNs per track.

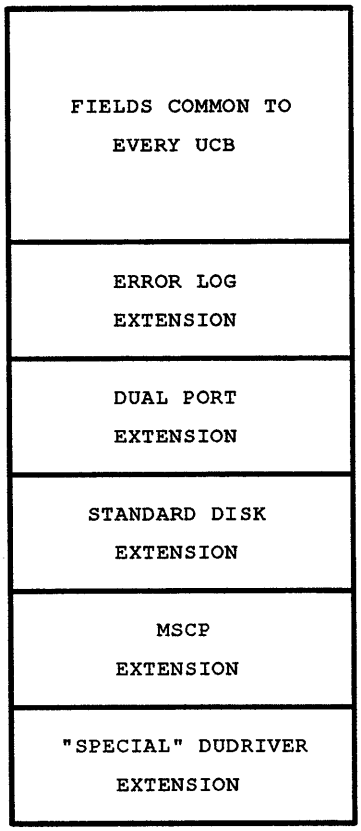
  — Number of tracks per group.

  — Number of groups per cylinder.

  SDA currently does not display this information pertaining to this extension with the UCB format command, but the symbolic offsets are globaly available.

Figure 2–5 illustrates the general format of a UCB representing a DSA disk, or a disk which is MSCP-served by its remote VAX host.

Figure 2–5: UCB Extensions for MSCP Served Disk



2.2.4 CDDB - Class Driver Data Block

A class driver data block contains parameter and status information specific to an "MSCP speaking" controller and its MSCP server. It also keeps a queue of active MSCP commands issued to that controller but for which no corresponding end messages have been received. Some of the specific items of information kept in a CDDB are:

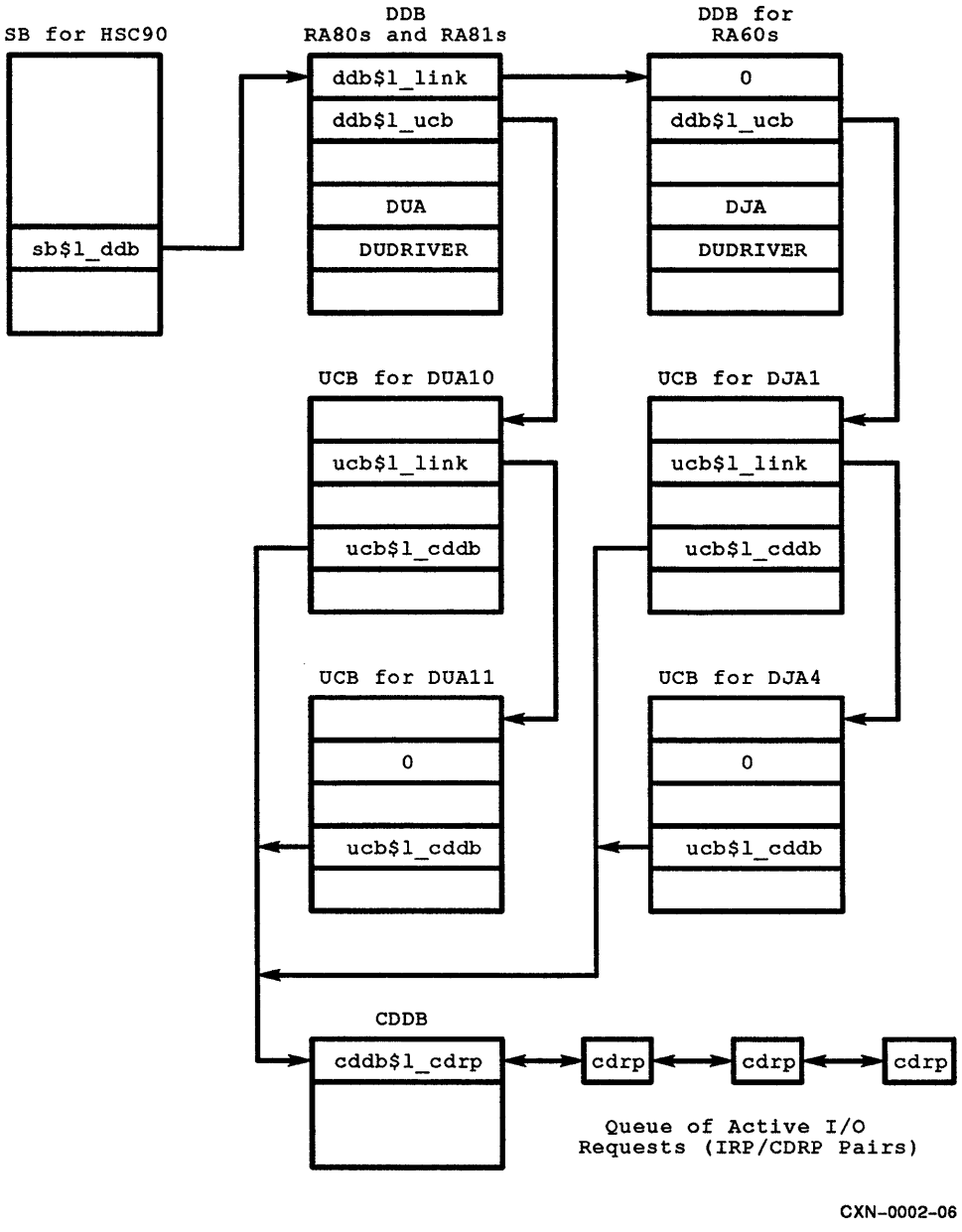
- Queue of active MSCP commands (CDRP/IRP pairs) issued to the controller but for which corresponding MSCP end messages have not yet been received.
- Whether or not controller is making progress with oldest active command.

- Controller's class, model, unique device number, and system ID.
- Controller flags such as if the controller supports volume shadowing, if it supports disks formatted with 576-byte sectors, and if it handles bad block replacement by itself.
- Status flags related to the controller's MSCP server, such as:
  - — Currently no connection exists between local disk class driver and controller's MSCP server.
  - — Connection between the local disk class driver and the controller's MSCP server is being initialized.
  - — Local disk class driver is reconnecting to controller's MSCP server.
  - — Connection closed for Port Load Balancing
  - — Local host is currently polling controller's MSCP server for units to determine what units the local host should include in its I/O database.
  - — MSCP server is being handed I/O requests by the local host in single stream mode.
- Allocation class of the controller (or 0 if none).

2.2.4.1 Linkage From UCBs to CDDB for Controller

Figure 2–6 illustrates that each UCB contains the address of the CDDB corresponding to the DSA controller currently handling its I/O requests. This address is stored at UCB offset *UCB$L_CDDB*.

Figure 2–6: CDDB Linkage Maintained by each UCB



If a remote VAX is running the VMS based MSCP server and is serving disks to the VAXcluster, then that remote VAX appears to the local VAX as a "logical DSA controller". The local disk class driver issues MSCP commands to the remote VAX's VMS based MSCP server in exactly the same way that it would to the server on an HSC.

The preceding diagram applies to such a situation. The SB in the diagram would actually be for the remote VAX rather than a DSA controller. The UCBs could be for any type of disk on the remote VAX that have been "set served". And the CDDB would play the same role for the remote "logical DSA controller" as it would for an HSC.

The preceding diagram also applies if the DSA controller is local (such as a KDM70 on a local VAX.) However, remember that while there is an SB representing the local DSA controller, the DDBs are not linked to that SB. DDBs for the local DSA controller are linked to the permanent local SB instead.

2.2.4.2 Linkage from CDDB to UCBs on that Controller

By means of its UCB$L_CDDB field, each UCB keeps track of the particular CDDB for the controller currently handling its I/O requests. However, it is also necessary for the CDDB to keep track of all UCBs that can give it I/O requests. This is facilitated by a linked list. At offset CDDB$L_UCBCHAIN is the address of the first UCB in this list. Then, within each UCB, offset UCB$L_CDDB_LINK provides the address of the next UCB in the list. The list ends with a UCB whose CDDB_LINK field contains a 0.

This linkage proves useful when DUDRIVER needs to determine if a particular disk is already known to be on a controller. It merely fetches the CDDB corresponding to the controller and then scans this list looking for a matching UCB. (This also applies to the case where the "controller" is actually a VAX running the VMS based MSCP server.)

2.2.4.3 Linkage from CDDB to DDBs on that Controller

All DDBs for disks handled by a DSA controller are kept in a singly linked list. The address of the first DDB in this list is kept at offset *CDDB$L_DDB* in the CDDB associated with the controller. Each DDB then contains the address of the next DDB in this list at offset *DDB$L_CONLINK*. A DDB whose CONLINK field is set to 0 represents the last DDB in the list.

One application for this list occurs when a UCB is created for a newly discovered unit on a DSA controller which will provide the primary path for the unit. The list of DDBs attached to the CDDB is searched for one with a matching generic name (DUA, DJA, ...). If one is found, the new UCB is linked into that DDB's list of UCBs. If a matching DDB is not found, it is created and then the new UCB becomes the first in the DDB's list of UCBs. (Again, this also applies to CDDBs associated with a disk on a remote VAX acting as a "logical controller" by running the VMS based MSCP server.)

2.2.4.4 Extensions to Disk Class Driver CDDB

There are two extensions to the disk class driver CDDB. The first is the *"permanent IRP/CDRP"*, and the second is the *"DAP IRP/CDRP"*.

The permanent IRP/CDRP is used by DUDRIVER's timeout mechanism for two purposes:

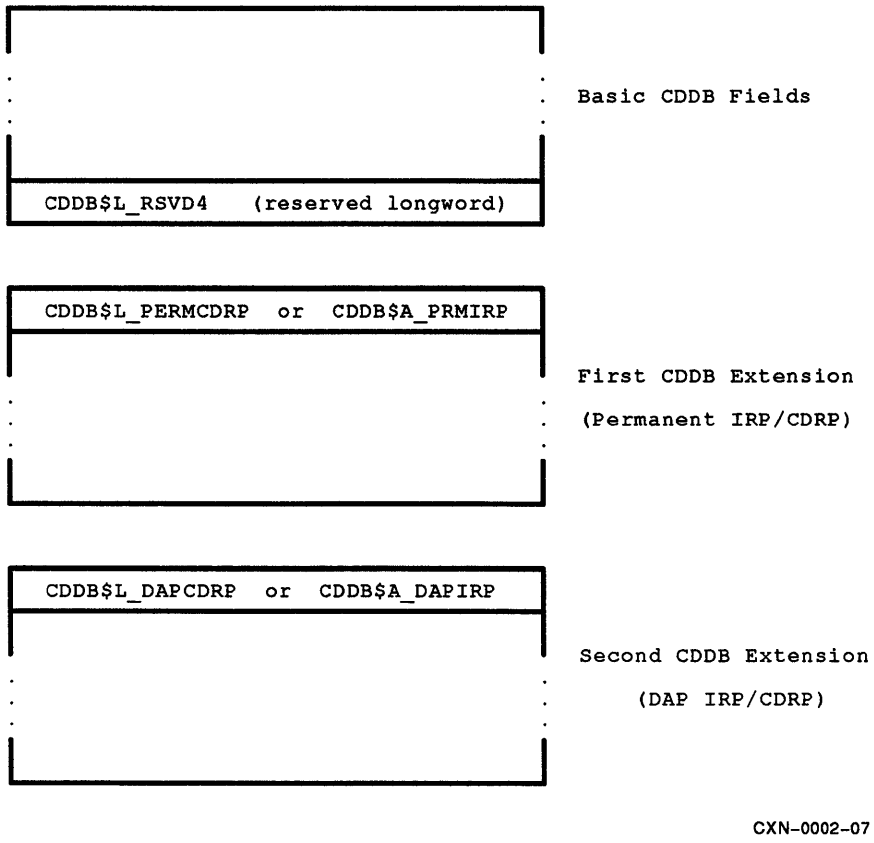
- During periods of inactivity, DUDRIVER will issue what effectively amounts to a NOP to the controller so that it knows the local host is still "alive and well".
- If the oldest active MSCP command to a controller has been pending for an "excessively long period of time", DUDRIVER issues a GET COMMAND STATUS to the controller to see if any progress has been made on that command.

The DAP IRP/CDRP is used by DUDRIVER for Determine Access Paths processing. This is a mechanism for finding out if a unit is dual pathed between two DSA controllers (e.g. an RA81 statically dual pathed between two HSC90s).

See the discussions on DUDRIVER's timeout mechanism and Determine Access Paths processing elsewhere in this book for details.

Figure 2–7 illustrates the general format of a CDDB used by the disk class driver., This includes the two special class driver extensions. Symbolic offset names relative to the beginning of the CDDB indicate where the basic CDDB ends and where each of the two extensions begin.

Figure 2–7: CDDB Format and Class Driver Extensions



2.2.5 CRB - Channel Request Block

Within a generic *channel request block* is the head of a queue of waiting fork blocks, each of which represents the suspended context of a driver fork process waiting to gain control of a controller data channel. A generic CRB also holds the addresses of entry points for *driver interrupt service routines* as well as device and controller initialization routines.

While port drivers make use of these generic CRB fields, DUDRIVER is essentially unconcerned with them. This is because no device directly interacts with or interrupts DUDRIVER. The disk class driver exchanges information with the controllers for its disks through the SCS and port driver layers of software between it and the controllers.

DUDRIVER still creates a CRB for each controller to which it "talks". The purpose of each such CRB is to trigger periodic tasks which DUDRIVER needs to perform for the controller associated with the CRB. These tasks include:

- Determining if progress has been made on the oldest active command issued to that controller and taking appropriate action.
- If no commands are active, issuing a NOP to the controller so that it knows that DUDRIVER on this host is still "alive and well".
- Invoking DAP (determine access paths) processing to find secondary paths to dual-ported disks.

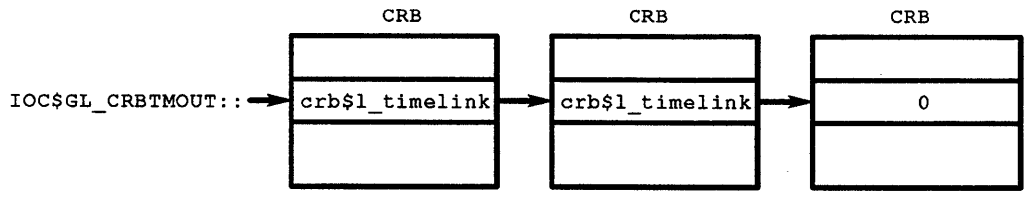
Thus, there are three fields in a CRB that are of principal interest to DUDRIVER:

- DUETIME

  All CRBs are kept in a list whose head is at location *IOC$GL_CRBTMOUT-2C*. Scanning the list for timed out CRBs is one of the "once a second" tasks performed by routine *EXE$TIMEOUT* in module TIMESCHDL. A CRB's *DUETIME field* (crb$l_duetime) contains the time in seconds when the CRB will time out. If the content of this field is less than or equal to the content of *EXE$GL_ABSTIM*, then the CRB has timed out and the routine to perform periodic tasks associated with this CRB is called.

  Figure 2–8 illustrates the CRB timeout list. Location IOC$GL_CRBTMOUT contains the address of the field CRB$L_TIMELINK in the first CRB in the list. The CRB$L_ TIMELINK field of each CRB contains the address of the next CRB timelink field in the list. A zero value in the CRB$L_TIMELINK field terminates the list.

Figure 2–8: CRB Timeout Linkage



NOTE

All CRBs created by all drivers are in this same CRB timeout list, and not just CRBs created by DUDRIVER.

- TOUTROUT

  The *TOUTROUT field* of a CRB contains the address of the routine to be called when the CRB times out. This routine performs the periodic tasks associated with the CRB. For DUDRIVER, these are the tasks described on the previous page.

The CRB's timeout routine should also reset the DUETIME field to reflect the next wakeup time for itself. For DUDRIVER, this is done by merely adding to the current time the "controller delta" stored in the *CNTRLTMO* field of the CDDB associated with the controller.

- AUXSTRUC

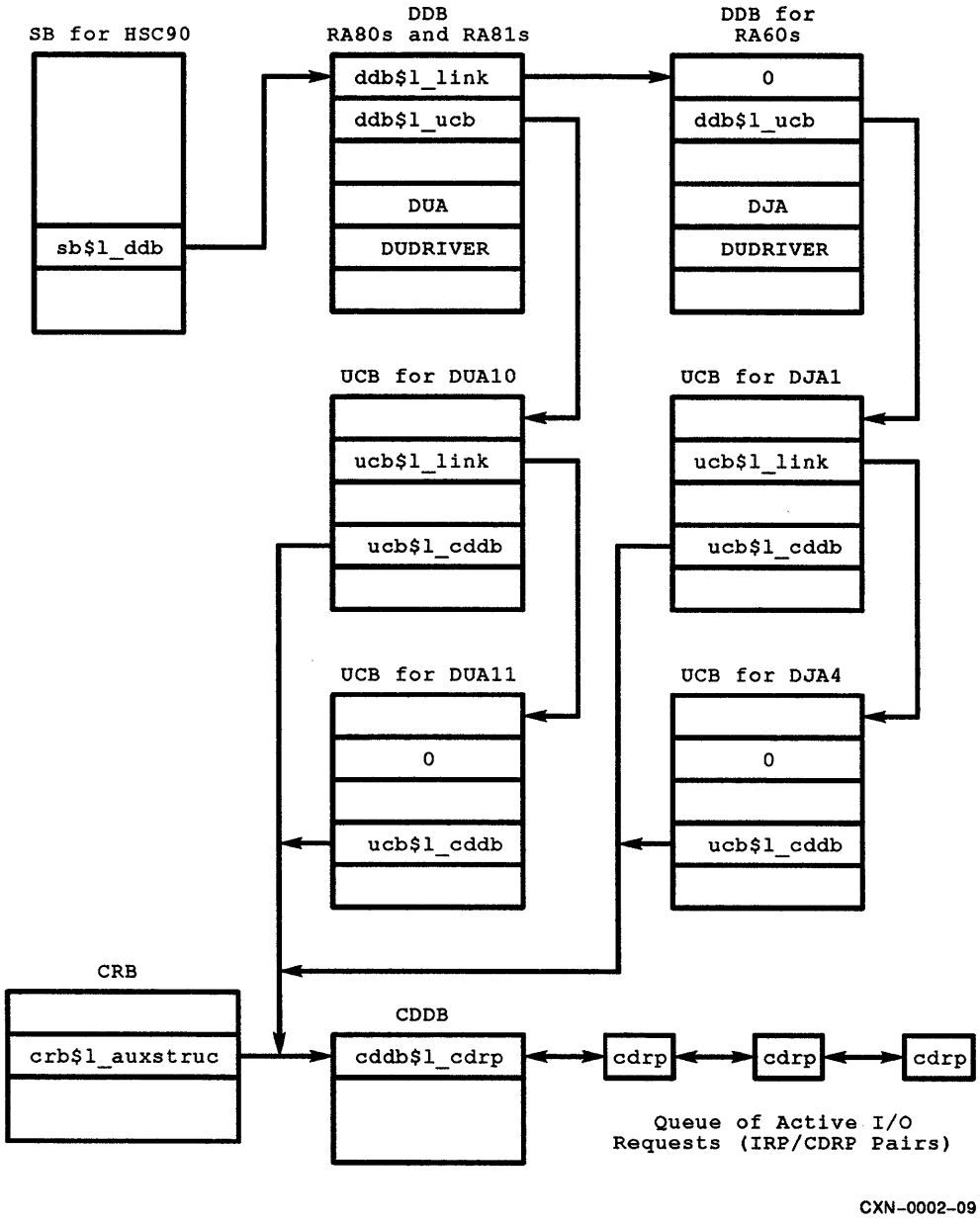
  This field contains the address of an *"auxiliary structure"* to be passed to the routine whose address is in the TOUTROUT field.

  DUDRIVER has one common timeout routine for all controllers, namely *DU$TMR*. It is therefore necessary to identify to that routine the particular controller for which it is being called. This is accomplished by having the CDDB associated with the controller be the "auxiliary structure" for the CRB also associated with that controller.

  When DUDRIVER creates and inserts a CRB for the controller into the IOC$GL_CRBTMOUT list, it stores the address of the controller's CDDB in the *CRB$L_AUXSTRUC* offset of the CRB.

Figure 2–9 illustrates the relationship of the CRB to the DUDRIVER I/O database diagram which has been evolving in this chapter.

Figure 2–9: CRB Linkage and the General Related Data Structures



2.2.6 Dual-Pathed Disks

Thusfar in this chapter, data structures have been presented on the assumption that there is only one controller for each disk. However, it is often desirable to port a disk to two different controllers, thus providing two different paths to that disk. The objective in doing so is to avoid the situation wherein a controller is a single point of failure for software needing that device.

No new data structures are needed to handle dual-pathed disks. In fact, the impact on the previously introduced data structures involves merely a few more fields in those structures.

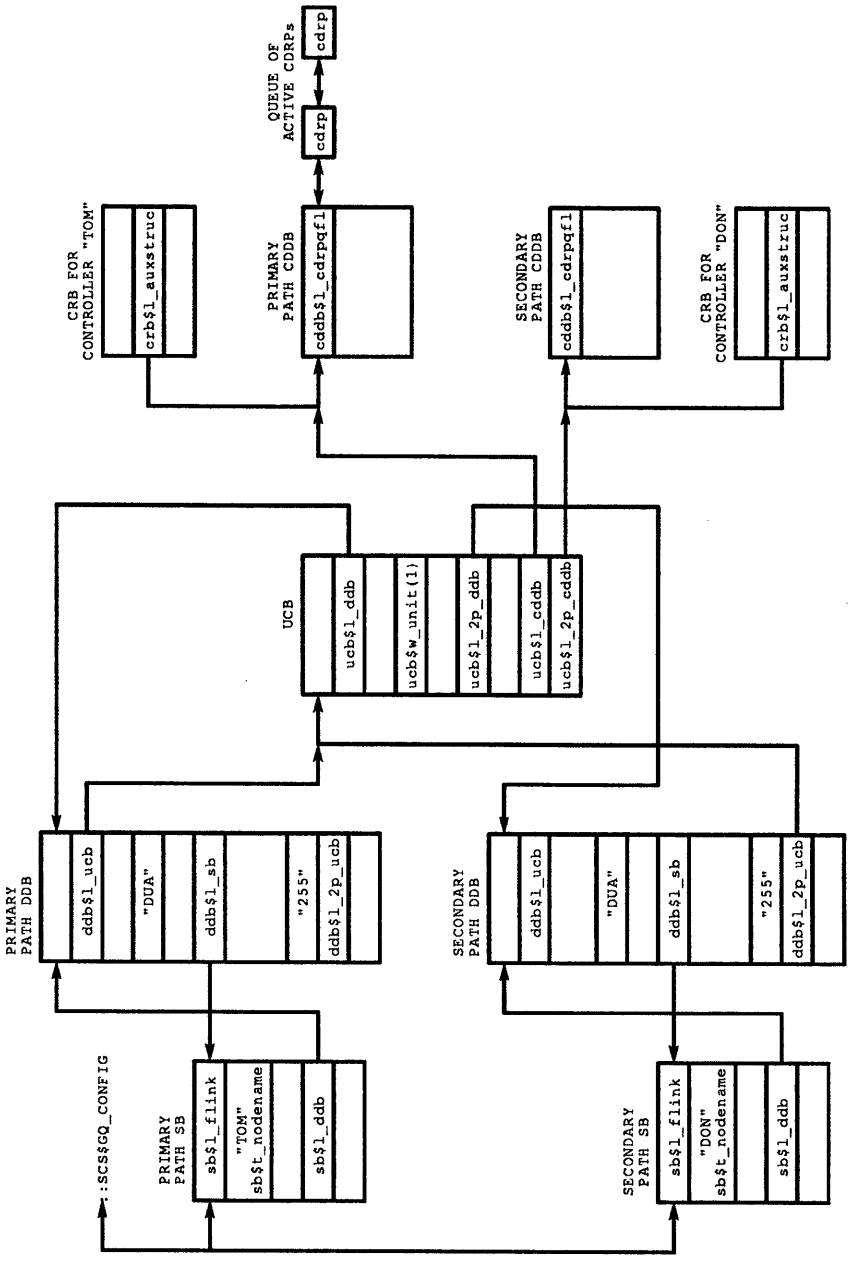
The following diagram is a simplified illustration of data structure linkages for a disk, (DUA1), dual-pathed between two controllers named TOM and DON. Controllers TOM and DON are in allocation class 255. Note the following points in this illustration that facilitate dual-pathing:

- There are two DDBs for the generic class of disks "DUA". One of them is in the SB$L_DDB list for controller TOM, and the other is in the SB$L_DDB list for controller DON. The DDB$L_NAME fields for both DDBs contain the generic "DUA". The DDB$L_ALLOCLS field for both contain the quantity "255".
- Both DDBs for generic DUA "point" to a UCB whose UCB$W_UNIT field contains "1". Hence, the disk is named DUA1 regardless of which controller is used to access the disk.

  However, the local VAX will direct its I/O to the disk using only the "primary path". The "secondary path" exists strictly for failover of the disk in the event that the primary path to the disk is lost for some reason. Since the primary path for the disk is controller TOM, the UCB for the disk is in the DDB$L_UCB list for controller TOM. But since controller DON provides the secondary path, the UCB for this disk is in the DDB$L_2P_UCB list for DON's DDB.
- Within the UCB, the DDB field contains the address of the primary path DDB, and the 2P_DDB field contains the address of the secondary path DDB.
- Within the UCB are pointers to two CDDBs. The UCB$L_CDDB field contains the address of the primary path CDDB, whereas the UCB$L_2P_CDDB field contains the address of the secondary path CDDB.

  All CDRPs representing I/O requests for this unit are queued to the primary path CDDB. It will be seen in the chapter presenting the detailed flow of a $QIO that the primary path is always selected by the disk class driver. Figure 2–10 shows the data structures involved for secondary paths.

Figure 2–10: Data Structures Supporting Secondary Paths



As has already been pointed out, the diagram on the previous page is somewhat simplified. Here is an explanation of those simplifications.

Only the two system blocks corresponding to HSC controllers TOM and DON are shown to be in the SCS$GQ_CONFIG queue. The first entry in this queue would actually be the local system's SB, and the SBs for TOM and DON would be linked off of that entry.

There would also be system block entries for all other nodes, such as other VAXes and HSCs with which the local VAX communicates via SCS. There would be still more SBs for local DSA controllers, such as a local KDM70.

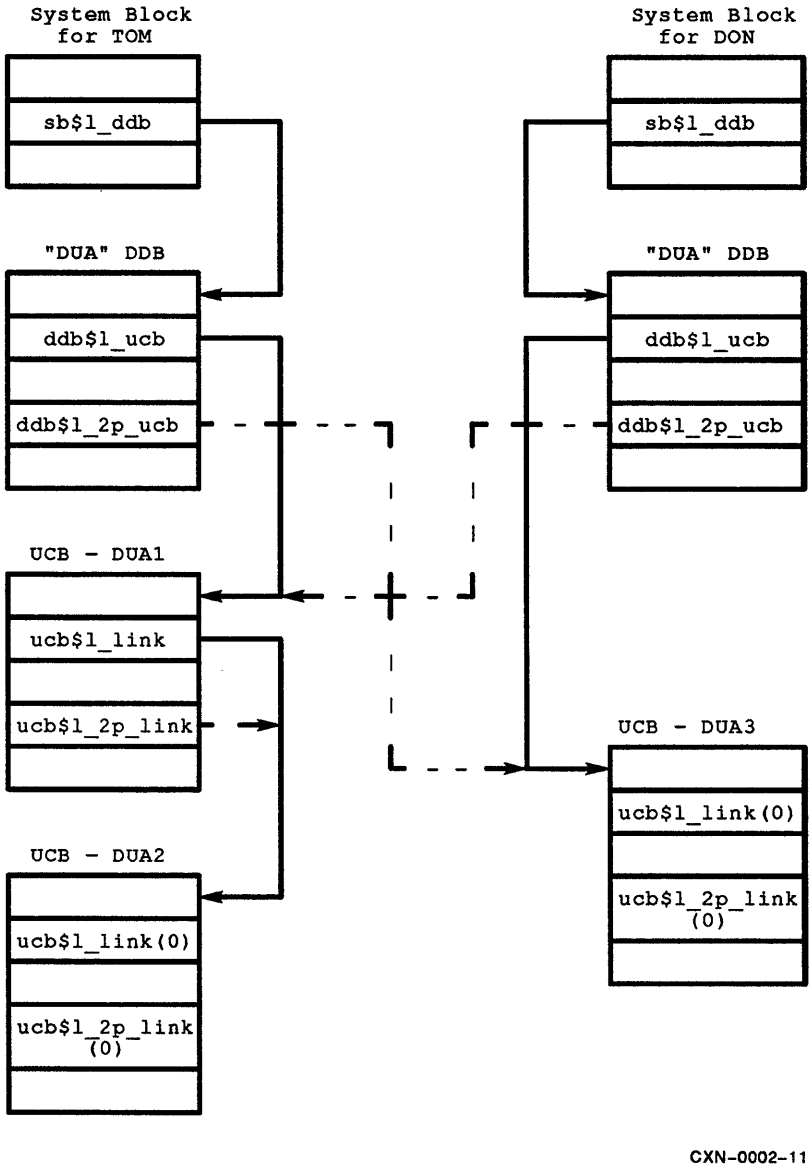
The next simplification is that only one DDB is shown to be present in each system block's SB$L_DDB list. In fact, these DDBs may merely be the first DDBs in these lists. For example, if controller TOM had disks called DJA7 and DJA8, then the DDB$L_LINK field of TOM's DUA DDB would contain the address of another DDB with generic name "DJA".

What if there were more DUA disks for which controller TOM were providing the primary path? In this case, the UCB shown would be only one of several in a list linked to TOM's DDB. The DDB$L_UCB field of controller TOM's DDB would contain the address of the first UCB in this list. The *UCB$L_LINK* field of each UCB in this list would contain the address of the next UCB in this primary path list. A zero value in the UCB$L_LINK field would terminate the list.

Similarly, if controller DON were providing the secondary path to more than just one DUA disk, then there would be a secondary path list of UCBs linked to DON's DDB. The DDB$L_2P_UCB field of DON's DDB would contain the address of the first UCB in this list. The UCB$L_2P_LINK field of each UCB in this list would contain the address of the next UCB in the list. The UCB$L_2P_LINK field is not part of the basic UCB, but is found in the dual path extension.

It is even permissible for a controller to provide the primary path for some disks and provide the secondary path to other disks. Figure 2–11 shows a simplified illustration of how controller TOM provides the primary path for DUA1 and DUA2, and the secondary path for DUA3. DON, on the other hand, provides the secondary path for DUA1 and DUA2, but the primary path for DUA3.

Figure 2–11: Provisions for Secondary Paths Offered by Multiple Servers



2.3 DUDRIVER I/O Database Initialization

The disk class driver's database consists of CDDBs and CRBs associated with "MSCP speaking" controllers, and the DDBs and UCBs associated with the disks on those controllers. In general, DUDRIVER initializes these data structures within the context of two general scenarios:

- Controller Initialization.

  Once VMS has become aware of the existence of an "MSCP speaking" controller, it establishes an SCS connection with the MSCP disk server (MSCP$DISK) on that controller. Based on the information exchanged between the disk class driver and the server, a CDDB and CRB are initialized. Then the class driver queries the server regarding disks and builds UCBs and DDBs corresponding to them. As a result of the dialogue that occurs during controller initialization, the server is set to a *"controller online"* state from the class driver's point of view.

- Attention Messages Received by DUDRIVER from MSCP$DISK.

  An MSCP server sends an *AVAILABLE ATTENTION* message to a disk class driver which it considers "controller online" anytime a unit asynchronously becomes available to that class driver. An *ACCESS PATHS ATTENTION* message is used by an MSCP server to report an alternate (i.e. secondary) access path for a disk. When DUDRIVER receives such messages, it builds new UCBs to reflect newly discovered units, or alters existing UCBs to reflect secondary paths to already known units.

  A third type of attention message, *DUPLICATE UNIT NUMBER*, is used to notify hosts that two or more units have conflicting unit numbers so that an operator can take appropriate action.

The next few sections of this chapter are concerned with when these scenarios occur, and what happens during each.

2.3.1 DUDRIVER's Controller Initialization Routine

2.3.1.1 DU_CONTROLLER_INIT

DU_CONTROLLER_INIT is DUDRIVER's top level controller initialization routine. It is called whenever it is necessary to establish an SCS connection with the MSCP disk server on an "MSCP speaking" controller. It then proceeds to modify the local host's I/O database to reflect disk units accessible to this host through the server in that controller. This can typically happen under any of four circumstances which are described in this section and illustrated by the flowchart which follows. The following list indicates some of these circumstances:

- System disk for local host is on an "MSCP speaking" controller.

  VMS initialization begins when SYSBOOT transfers control to module INIT. At location *INI_BOOTDEVIC*, INIT performs the allocation and initialization of the database to describe the system disk.

If the system disk is handled by a DSA controller, then SYSBOOT has loaded the port driver and disk class driver into nonpaged pool at locations *BOO$GL_PRTDRV* and *BOO$GL_DSKDRV*, respectively.

INI_BOOTDEVIC calls routine *IOC$INITDRV* twice: once to initialize the port driver (which also causes port microcode to be loaded and started), and the second time to initialize the disk class driver. It is this second call to IOC$INITDRV that invokes DU_CONTROLLER_INIT. It should be observed, however, that the calling of IOC$INITDRV twice here pertains only to the controller handling the system disk, and no other controller.

NOTE

If this host has been booted from a remote system (i.e. this host is participating in a VAXcluster via the ETHERNET), then IOC$INITDRV will also allocate and initialize a System Block for the remote system.

- Local Host Participates in a VAXcluster.

  When INIT completes its work, it transfers control to the scheduler which selects the swapper to run. The first time the swapper runs, it creates the SYSINIT process. At location *SIP_CLUSTER_INIT*, the SYSINIT process tests to see if the local host is going to participate in a cluster. If this test proves true, SYSINIT creates the *"stand alone configure"* process (STACONFIG). STACONFIG calls BOO$CONFIGALL to autoconfigure all local adapters and devices specified by a list it passes to the routine.

  — If the local host does not have NI cluster potential, (*NISCS_LOAD_PEA0* sysgen parameter equals zero), then the list includes only devices (and hence drivers) beginning with the letters D, P and M[1].

  — If the local host does have the potential to use the NI for cluster communication (NISCS_LOAD_PEA0 nonzero), then the list includes all devices (and hence drivers) beginning with the letters X, E and F[2]as well as D, P and M.

  NOTE

  If the local host booted from a local disk, then the NI or CI port driver would not have been started (and hence port microcode not loaded and started) by INIT. However, here is where that task would be done in such a case if the local VAX is going to participate in a cluster.

Within BOO$CONFIGALL, a call is made to IOGEN$LOADER to load the database and driver if necessary. And from within IOGEN$LOADER, DU_CONTROLLER_INIT would be called.

[1] Device code for TAPE drivers has been included in VMS V5.4-3

[2] Device code for FDDI drivers has been included in VMS V5.4-3

NOTE

If the SYSGEN parameters for the local host indicate that a quorum disk is being used, then STACONFIG also starts fork threads to autoconfigure MSCP- and HSC-served disks so that the quorum disk can be found.

- "Autoconfigure All" in STARTUP.COM .

  SYSINIT eventually invokes the STARTUP.COM procedure. Among the many operations performed by this procedure is to perform an *"autoconfigure all"* for local adapters and devices. Thus, even if the stand alone configure process was not created, SCS port drivers and their associated ports would be loaded and started. This would also subject local DSA controllers to the processing done by DU_CONTROLLER_INIT if not already done by STACONFIG.

- Remote "MSCP Speaking" Controllers Discovered by CONFIGURE.

  After doing the "autoconfigure all", STARTUP.COM then creates the *CONFIGURE process* if either the local host is participating in a VAXcluster or at least one of the following ports are present (CI, DSSI, or a local DSA port).

  The role of the CONFIGURE process is to discover remote DSA controllers which have not as yet been found. This applies both to HSCs and remote VAXes running the VMS based MSCP server. Once found, these controllers will also be subjected to local DU_ CONTROLLER_INIT processing.

  CONFIGURE performs its remote MSCP Server locating by establishing a periodic polling mechanism with the assistance of SCS routines. Thus, anytime an HSC makes an unexpected appearance on the CI, or a DSSI based device becomes known, it will be seen and subjected to DU_CONTROLLER_INIT by CONFIGURE. This also applies to a remote VAX on the CI or NI running the VMS based MSCP server. Figure 2–12 and Figure 2–13 depict the flow of device configuration.

Figure 2–12: Configuration of Devices by Sysboot and Init

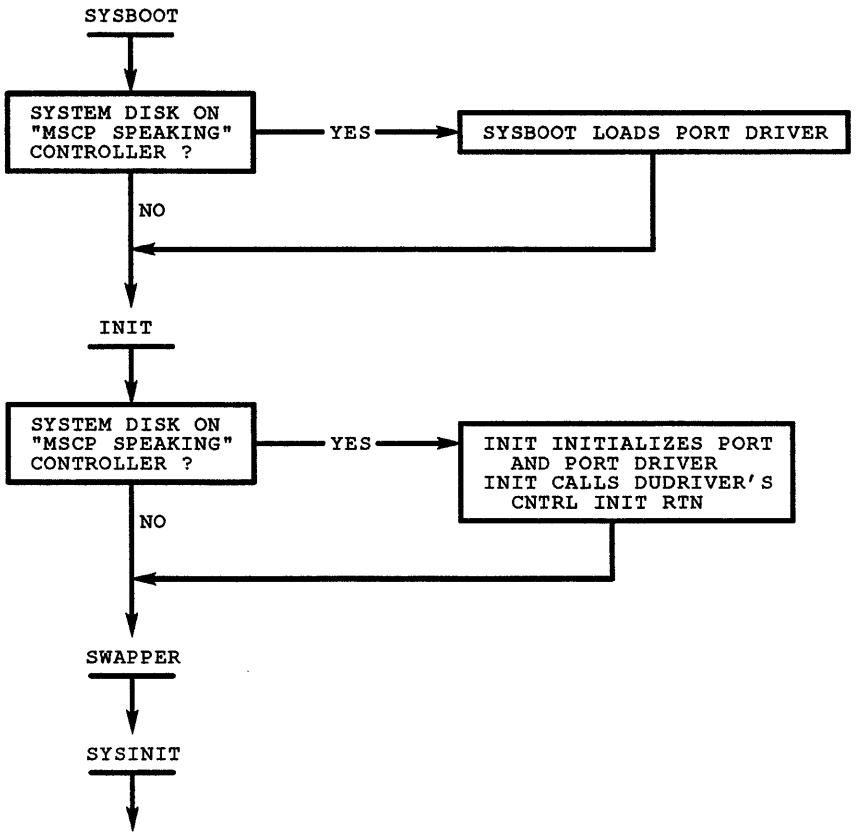
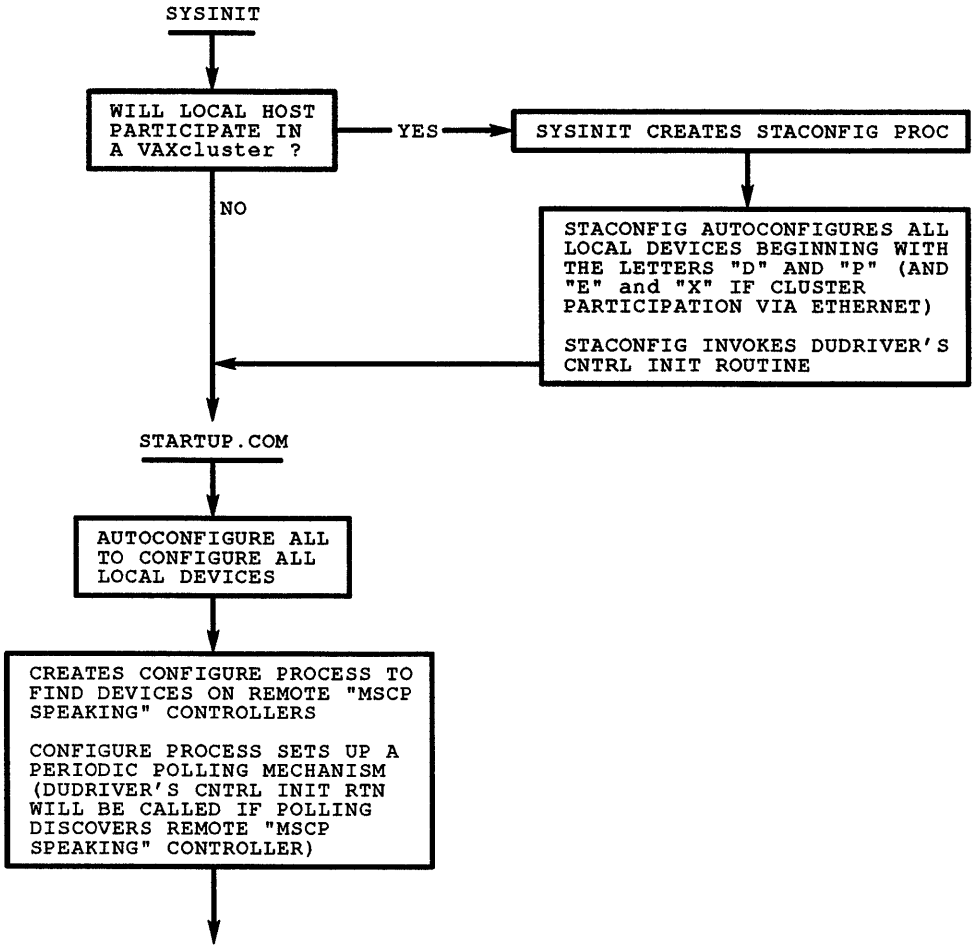


Figure 2–13: Configuration of Devices by Sysinit and Startup



2.3.2 Overview of DUDRIVER's Controller Initialization Routine

2.3.2.1 CDDB Creation and Initialization

First, DU_CONTROLLER_INIT allocates and initializes a CDDB for the controller. The controller's system ID is copied into the CDDB. Among the flags which get set here are the INITING and NOCONN flags. These indicate that the CDDB is being initialized and that there is as yet no SCS connection between the local disk class driver and the controller's MSCP server.

2.3.2.2 MAKE_CONNECTION Establishes a Connection to MSCP Server

Next, DU_CONTROLLER_INIT calls *MAKE_CONNECTION* to establish an SCS connection with the MSCP server. At this time, the addresses of routines within DUDRIVER for receiving datagrams and messages from the controller are declared. And various controller characteristics such as its timeout period, controller flags, software and hardware versions, and allocation class are established and recorded in the CDDB.

Upon return from MAKE_CONNECTION, DU_CONTROLLER_INIT sets up the timeout routine address and duetime fields in the CRB associated with the controller.

2.3.2.3 Poll for Disk Units

It then calls *DUTU$POLL_FOR_UNITS* to query the controller's MSCP$DISK about disks which it may access. The polling mechanism is "conceptually" quite simple. A series of *GET UNIT STATUS* commands (each with the "next unit" flag set) is issued to the server.

A UCB will be setup by one of the two following routines based on the content of the end messages corresponding to the commands.

- Routine *DUTU$NEW_UNIT* sets up the UCB if the disk reported in the end message is not the system disk.
- Routine *DO_ORIG_UCB* sets up the UCB if the disk is the system disk. This routine will complete filling in the original UCB created by the INIT process.

2.3.2.4 Check for Controller Based Shadow Set

If the Restart Parameter Block indicates that the system disk is part of a CONTROLLER based shadow set, DO_ORIG_UCB creates a shadow set virtual unit consisting of this one member. (The remaining members of the system disk shadow set virtual unit should be added by a MOUNT/SHADOW command in the SYSTARTUP_V5.COM procedure.) DDBs will also be created as needed. For HOST based shadow sets, this would have been performed when we configured the secondary class driver in Init.

At the end of the polling procedure, units which have never been seen before will have their UCBs linked into the database such that the controller just polled will be the primary path for these units. Such would be the case for single ported units, and units which are statically dual-ported.

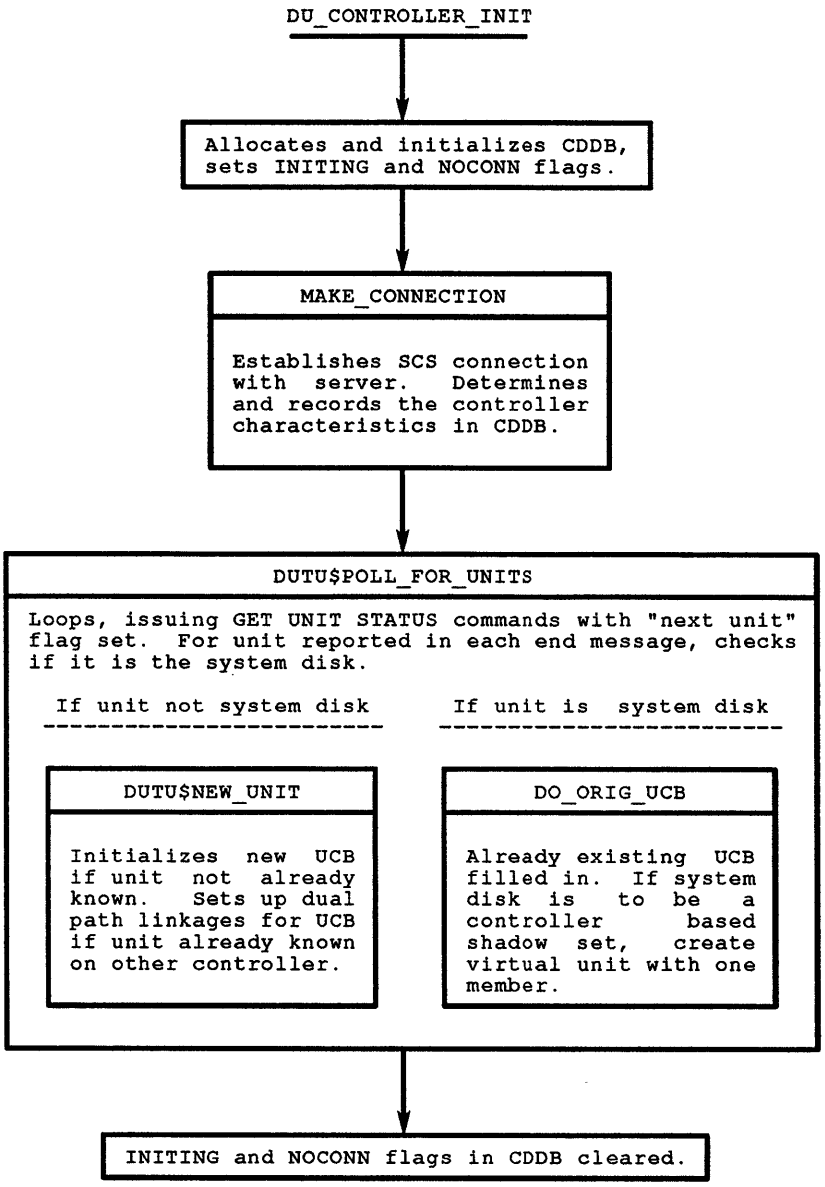
2.3.2.5 Handling of Secondary Path Discovery

If the unit is dynamically dual-ported, then polling may find a secondary path to an already known unit. While there is some special casing here (see the detailed routine description), in general this latter case will result in an already existing UCB having its secondary path linkages setup. However, secondary path linkages for shadow set virtual unit UCBs are not setup by this procedure.

Finally, the INITING and NOCONN flags in the CDDB are cleared.

Figure 2–14 illustrates the general flow of the events that occur in DU_CONTROLLER_INIT.

Figure 2–14: DU_CONTROLLER_INIT flow



2.3.3 Determine Access Paths Processing

2.3.3.1 Determination of Topology of Disk Units

Determine access paths (DAP) processing is used by class drivers to determine the topology of units that are ported to more than one controller.

2.3.3.2 Access Path Attention Messages

For the disk class driver, DAP processing involves issuing a Determine Access Paths MSCP command for each unit on the controller whose UCB indicates that it is online and that it is not a shadow set virtual unit. Upon receipt of the DAP command, the unit will identify itself to any other controller to which it is connected; and that other controller's MSCP server will then issue *Access Path Attention messages* to all disk class drivers to which it is "controller online".

Observe that in this situation, only the first controller knows who issued the Determine Access Paths command, and not the second. Therefore, the second controller must send the Access Path Attention messages to all disk class drivers to which it is "controller online".

Because of this, the local DUDRIVER may receive an unsolicited Access Path Attention message. In fact, there is the possibility that the local DUDRIVER may receive an Access Path Attention message for a unit it does not yet even know about (i.e. that it has not yet discovered and for which it has not yet set up a primary path).

2.3.3.3 Setup of Dual Path if Found

Upon receipt of an Access Path Attention message from any controller for any unit, DUDRIVER will call routine *DUTU$SETUP_DUAL_PATH* to search out the UCB corresponding to the unit reported in the message. If DUTU$SETUP_DUAL_PATH finds the UCB, then it sets up the appropriate dual-path linkage for it. If the UCB is not found, the message is merely ignored. (For further details, see the detailed description of routine ATTN_MSG near the end of this chapter. Also, an overview of routine DUTU$SETUP_DUAL_PATH is presented in the detailed description of routine DUTU$NEW_UNIT.)

2.3.3.4 DAP Scheduling

DAP processing is performed by routine *DUTU$DODAP* in DUDRIVER for non emulated MSCP servers. It is invoked as part of the disk class driver timeout mechanism driven by routine DU$TMR. Periodically, the CRB associated with a particular controller times out and invokes DU$TMR. Routine DU$TMR invokes DAP processing if either of two conditions is true:

- It finds that there are no MSCP commands active for that controller.

- The oldest active command has been around a "very long time"; but, nevertheless, the controller is "making progress" on this command. (Details of DU$TMR and a detailed description of this routine are presented in a later chapter of this book dealing with DUDRIVER error handling.)

2.3.3.4.1 DAPBSY Flag Set in the CDDB if DAP Processing in Progress

DAP processing is, in fact, not done every time DUTU$DODAP is called. If DUTU$DODAP actually initiates DAP processing, the *DAPBSY* flag is set in the CDDB associated with the controller. The address of this CDDB is in the AUXSTRUC field of the CRB associated with the controller. The DAPBSY flag is not cleared until DAP commands have been issued for all UCBs which are online and do not represent shadow set virtual units.

2.3.3.4.2 DAPBSY Flag Checked for DAP Already in Progress

Since each invocation of DUTU$DODAP is actually a fork thread, it is possible under certain very heavy load situations that a previous DAP processing fork thread has not completed by the time a new fork thread is initiated. Consequently, if this flag is found to be still set, the new fork thread terminates itself without doing anything. Also, there is a DAPCOUNT field in the CDDB. Each time DUTU$DODAP is called, it decrements this field. If the field is still greater than or equal to zero, DAP processing is not done.

2.3.3.4.3 DAPCOUNT Field used to Determine Frequency of DAP Processing

The DAPBSY flag and DAPCOUNT field serve to strike a balance between the desirability of quickly finding secondary paths to units and the undesirability of inducing excess overhead and a negative performance impact by doing DAP processing too often.

NOTE

Each time the DAPCOUNT field in the CDDB becomes negative and DAP processing is therefore actually done, the DAPCOUNT field is reset to the value of the parameter *DAP_COUNT*. In VMS V5.5 the value of DAP_COUNT is hard coded as 11 (decimal) at the beginning of routine DUTU$DODAP.

One final observation should be made about DAP processing. This technique serves to detect new or changed alternate paths to a disk. However, the loss of a previously existing alternate path will not be detected.

2.3.4 Attention Messages

MSCP servers use *attention messages* to report certain asynchronous events relevant to a unit's availability and/or status to class drivers.

DUDRIVER's attention message processing routine, *ATTN_MSG*, dispatches based on the OPCODE field of the message buffer for the following three valid attention message types: Unit Available, Duplicate Unit, and Access Paths.

Otherwise, the message is considered invalid and logged as such, and the ATTN_MSG branches to *DU$RE_SYNCH* to reset what is assumed to be a "very ill" controller.

2.3.4.1 Unit Available Attention Message

ATTN_MSG dispatches to *UNIT_AVAILABLE_ATTN* to handle a unit available attention message. Routine DUTU$NEW_UNIT will either add a new unit to the local I/O database if the unit is not as of yet known, or it will alter the database to reflect the discovery of a secondary path to a known unit.

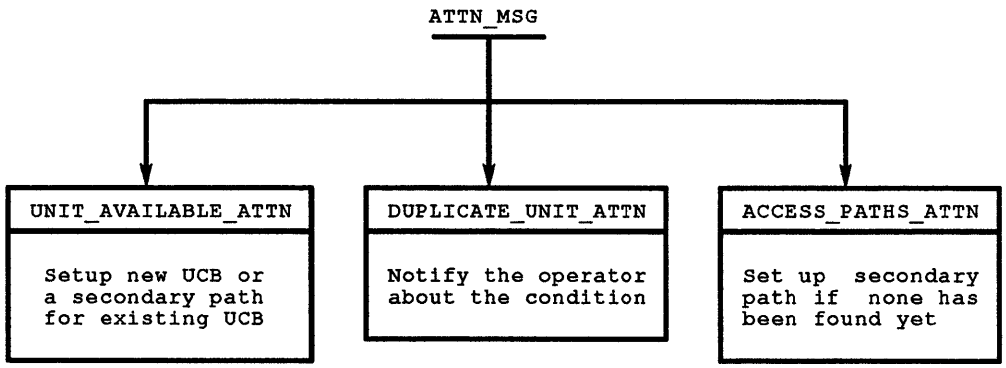
2.3.4.2 Duplicate Unit Attention Message

ATTN_MSG dispatches to *DUPLICATE_UNIT_ATTN* in the duplicate unit case. The only action taken upon receipt of a duplicate unit attention message is the notification of the operator about the discovery of two units on the same controller with the same MSCP unit number. The operator is then expected to resolve the situation.

2.3.4.3 Access Path Attention Message

ATTN_MSG dispatches to *ACCESS_PATH_ATTN* if an access paths attention message is received. This results in the calling of routine DUTU$SETUP_DUAL_PATH to alter the I/O database to reflect a secondary path to a known unit if that secondary path has not yet been found. (The flow of routine DUTU$SETUP_DUAL_PATH is presented within the detailed description of DUTU$NEW_UNIT.) Figure 2–15 illustrates the attention message dispatching.

Figure 2–15: Attention Message Dispatching



2.3.5 The CONFIGURE Process

The CONFIGURE process has the responsibility for discovering remote "MSCP speaking" controllers not found during the early stages of VMS initialization, and then subjecting them to DU_CONTROLLER_INIT processing.

Additionally, if the sysgen parameter *MSCP_SERVE_ALL* indicates that automatic disk serving is to be performed, the main routine of the configure process (*CONFIGMN*) establishes an executive mode timer AST to perform routine *AST_REC*. This routine executes every 15 seconds (VMS V5.5) and examines the local IO database using routine *SCAN_ALL_DEVICES* to determine if there are any new local devices that need to be MSCP served. If any are found, routine MSCP$ADDUNIT is called to perform the addition.

2.3.5.1 Configure uses SCS Process Polling to Discover MSCP Servers

To accomplish this, CONFIGURE requests the SCS process poller to seek out MSCP servers on remote nodes (HSCs, DSSI nodes and other VAXes). When one is found, the SCS process poller notifies the CONFIGURE process of this discovery, and also on what node the server is running. CONFIGURE then calls routine *BOO$CONNECT*, which in turn invokes DU_ CONTROLLER_INIT.

The CONFIGURE process consists of two major components: one to request the SCS process poller to poll for MSCP servers (disk and tape), and a second to handle the discovery of such servers.

2.3.5.2 Requesting Polling

Routine BOO$CONFIGURE requests SCS process polling. It does so in three steps:

- First, it creates a mailbox which is to be used by the SCS process poller to notify the CONFIGURE process when an MSCP server is found.
- Second, it calls routine REQ_POLL to request process polling by the SCS process poller. In so doing, BOO$CONFIGURE passes to the SCS process poller the I/O channel number of the mailbox, and also the names of the SYSAPs for which to poll.
- Finally, it uses the $QIO system service to create a "write attention" AST by which it will be notified anytime the mailbox is written (i.e. anytime a message is placed in the mailbox by the SCS process poller).

NOTE

The CONFIGURE process requests polling for both the disk and tape servers. However, only the disk server MSCP$DISK is of concern here.

The SCS process poller sends inquiries to what "appear to be logical remote controllers" from its point of view. This clearly includes actual remote controllers such as HSCs and ISEs. It also includes other VAXes in the cluster since, by means of the VMS based MSCP server code, these VAXes may appear as logical controllers.

If the SCS process poller receives a reply indicating that the desired server is "listening" for incoming CONNECT requests, then it delivers the write attention AST to the CONFIGURE process.

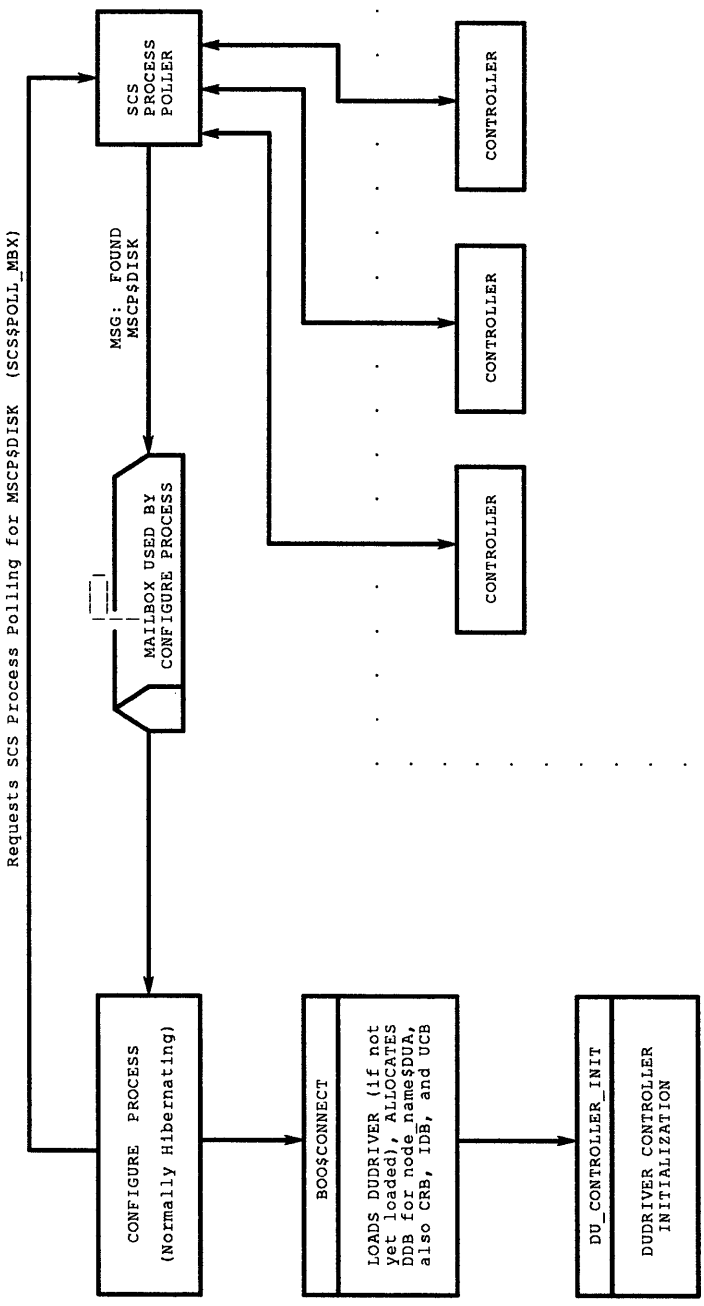
2.3.5.3 Discovery of MSCP Controllers

Writing to the mailbox triggers the delivery of the write attention AST. Routine *FOUND_PROC* in the CONFIGURE process is invoked to handle messages in the the mailbox. For each such message, FOUND_PROC calls BOO$CONNECT to do the following:

- Allocate a DDB corresponding to node_name$DUA .
- Initialize NAME, DRVNAME, SB, and ALLOCLS fields in DDB.
- Allocate and initialize a CRB for the controller.
- Allocate a UCB and initialize its UNIT field to 0.
- Call DU_CONTROLLER_INIT, passing it these data structures.

The interactions between the CONFIGURE process poller, the SCS process poller, and DU_CONTROLLER_INIT are illustrated by Figure 2–16.

Figure 2–16: Configure Process Polling and Device Configuration



CXN-0002-16

2.4 DUDRIVER I/O Database Initialization Routines

The remainder of this chapter presents detailed descriptions of the major routines involved in configuring the principal components of the VMS I/O database referenced by DUDRIVER.

DU_CONTROLLER_INIT	DUDRIVER's controller initialization. Invoked when it is necessary to add an "MSCP speaking" controller to the local node's database.
MAKE_CONNECTION	Called by DUDRIVER to establish a connection to the MSCP disk server in a controller. Also called during certain error handling when it is necessary to reconnect to a disk server.
DUTU$POLL_FOR_UNITS	After a connection has been made with an MSCP server, this routine queries the server for disks it is making accessible to the local VAX.
DUTU$NEW_UNIT	Invoked to search the I/O database to see if a unit is already known. If it is not, then this routine will add the unit into the database. If it is already known on some other controller, DUTU$NEW_UNIT will dual-path it if appropriate.
DUTU$DODAP	Issues Determine Access Paths commands to facilitate discovery of secondary paths for dual-ported disks.
ATTN_MSG	Processes incoming attention messages.

2.4.1 DU_CONTROLLER_INIT

2.4.1.1 Routine Process

Controller initialization begins with creating and initializing the CDDB to be associated with the controller. Next, it calls a routine to make a connection with the MSCP disk server in that controller. It then sets up the MSCP timeout mechanism. And finally, it polls the MSCP disk server for information with which it builds disk data structures within its I/O database.

- Routine DU_CONTROLLER_INIT sets the ONLINE flag in the UCB which it was passed.

NOTE

The UCB passed to DU_CONTROLLER_INIT may be for the boot device. If it is not, it will be deallocated shortly.

- The system id of the controller is stored in the *UCB$Q_UNIT_ID* field for later use in creating a CDDB corresponding to the controller.
- DU_CONTROLLER_INIT then creates a fork thread to perform the remainder of controller initialization at IPL$_SCS.

NOTE

This is done since numerous time consuming messages will be exchanged between this host and the remote controller. The number of tasks here increases with the number of disks on the controller.

- It calls *DUTU$CREATE_CDDB* to create and initialize a CDDB corresponding to controller. DUTU$CREATE_CDDB does the following:
  - — Allocates a CDDB from nonpaged pool and zeros it.
  - — Copies the controller's system id from the UCB into the SYSTEMID field of the CDDB.
  - — Sets the INITING and NOCONN flags in the STATUS field of the CDDB to indicate that it is initializing the CDDB, but also that a connection has not yet been established with the MSCP disk server in the controller.
  - — Sets ATTN (enable attention messages), MISC (enable miscellaneous error log entries) and THIS (enable this host's error log messages) flags in the CNTRLFLGS field of the CDDB for later use when setting controller characteristics.
  - — Initializes empty CDDB queues: CDRP, RSTRT, and CANCL.
  - — Initializes the failover control block within the CDDB.
  - — Initializes permanent and DAP CDRPs within CDDB by clearing each CDRP's RWCPTR field and setting each CDRP's PERM flag.
  - — Stores addresses of CRB and DDB into CDDB.
  - — Clears CONLINK field in the DDB.

— If UCB is not for the boot device, (i.e. if its VALID flag is not yet set), unlinks the UCB from the DDB by clearing the DDB$L_UCB field and then deallocates the UCB.

— If UCB is for the boot device (i.e. its VALID flag is already set and its address is already in location SYS$GL_BOOTUCB), the UCB address is stored in the ORIGUCB field of the CDDB.

— Searches chain of CDDBs, DU$DATA+DUTU$L_CDDB_LISTHEAD, to verify that there is no other CDDB with the same system id (i.e. to verify that this controller is not already known).

  o If some other CDDB with the same system id is already in the chain, then the CDDB just allocated would duplicate an already known controller's CDDB. Thus, this CDDB and associated UCB and CRB are deallocated so as not to make a second connection with the same MSCP$DISK, and this controller initialization fork thread terminates here without performing further work.

  o If no other CDDB with the same system id already exists in the chain, then the controller has just been discovered. So the newly allocated CDDB is inserted at the end of the CDDB chain and this controller initialization thread proceeds on.

— Calls IOC$THREADCRB to insert CRB on IOC$GL_CRBTMOUT list. CRB set to "infinite" timeout for now.

— Address of CDDB stored in AUXSTRUC field of CRB.

- DU_CONTROLLER_INIT calls MAKE_CONNECTION to establish an SCS connection with the MSCP disk server, MSCP$DISK, in the controller, and to determine the controller's characteristics.
- It tests the CDDB$W_CNTRLFLGS field to determine if the controller handles bad block replacement by itself. The MSCP$V_CF_REPLC bit will be set if so. If not, *DU$INIT_HIRT* is called to initialize the HIRT (Host Initiated Replacement Table) if the HIRT is not already initialized.
  - Clears HIRT's Replacement Request Queue (RPLQ).
  - Allocates permanent CDRP, RSPID, and MSCP command buffer to HIRT.
  - Allocates 4 pages of memory needed by replacement algorithm.
- The permanent timeout routine for this controller is established.
  - The CRB$L_TOUTROUT field is set to contain the address DU$TMR.
  - The CRB$L_DUETIME field is set to the current time plus the content of the CDDB$W_CNTRLTMO field, which was just set up by "set controller characteristics" transactions which occurred in MAKE_CONNECTION.

NOTE

The permanent CDRP in the CDDB is used only by routine DU$TMR.

DU$TMR always resets the DUETIME field in the CRB to the "then current" time plus the "controller delta" stored in the CNTRLTMO field of the CDDB.

- DUTU$POLL_FOR_UNITS is called to poll MSCP$DISK for units known to the controller and to alter the I/O database accordingly.

NOTE

DUTU$POLL_FOR_UNITS uses the DAP CDRP. The DAPBSY flag is set in the CDDB prior to the call to DUTU$POLL_FOR_UNITS so as to inhibit coincidental DAP processing requests for the controller from this host during polling.

- The NOCONN, DAPBSY, and INITING flags are cleared in the CDDB, and the controller initialization fork thread terminates itself.

2.4.2 MAKE_CONNECTION

2.4.2.1 Establishing a Connection

DUDRIVER establishes an SCS connection with the MSCP disk server in a controller. It then determines and records the controller's characteristics.

- Using the SYSAP name VMS$DISK_CL_DRVR, DUDRIVER uses the SCS service CONNECT to establish communication with SYSAP MSCP$DISK in the controller.

Routine	DUDRIVER Routine Name
MSGINP	DU$IDR
DGINP	DU$DGDR
ERRADR	DU$CONNECT_ERR

NOTE

If the CONNECT fails, the code pauses here retrying the CONNECT every CONNECT_DELTA seconds using a CRB-based timeout. (The CONNECT_ DELTA parameter is hard coded at the beginning of DUDRIVER to be 10 seconds in V5.5 of VMS.)

The *Path Move* bit in the CDDB status word indicating an entry resulting from a forced move request is cleared to allow the failover code to attempt to locate an alternate path.

— The address of the CDT associated with the connection is stored in the CDT field within the permanent CDRP in the CDDB, and also in DAPCDT field within DAP

CDRP in the CDDB. The address of the PDT corresponding the port supporting this connection is also stored in the CDDB.

- The MSCP command SET CONTROLLER CHARACTERISTICS is issued to MSCP$DISK.
  - — Allocates RSPID and message buffer in which to build command.
  - — Builds command.
    - o Host settable characteristics from CDDB$W_CNTRLFLGS inserted into command.
    - o Time from EXE$GQ_SYSTIME inserted into command.
    - o Host timeout temporarily set to "infinite" via this command.
  - — Sends command.
- The data from SET CONTROLLER CHARACTERISTICS end message received from controller is recorded in the CDDB:
  - — CNTRLFLGS.
  - — Sets/clears BSHADOW flag depending on whether or not controller supports volume shadowing.
  - — CNTRLTMO.
  - — CNTRLID.
  - — CSVRN (software and hardware version).
  - — MAXBCNT.
- The correct allocation class for the controller is stored in the CDDB and all DDBs currently linked to the CDDB.

NOTE

End message returns the allocation class (or 0 if none set) for controller.

During controller initialization, only one DDB should be linked to the CDDB at this time.

- The message buffer containing the end message is recycled, as is the RSPID.
- The correct host timeout interval is determined and set in the controller.
  - — For controllers which have dual path capability, it is computed as the larger of the value of the controller timeout just returned in the end message and the constant HOST_TIMEOUT as fixed in module DUDRIVER (30 seconds for VMS V5.5).

    Another SET CONTROLLER CHARACTERISTICS is issued to properly set the final host timeout in controller. The data from the associated end message is recorded in the CDDB and the RSPID and Message buffer are recylced.
  - — For controllers which do not have dual path capability the timeout field is set to an infinite value (zero).

2.4.3 DUTU$POLL_FOR_UNITS

Polling an MSCP server for units is performed by DU_CONTROLLER_INIT calling routine DUTU$POLL_FOR_UNITS after it has established a connection with the MSCP disk server within a controller. This operation is also invoked during error recovery after a connection has been re-established with the MSCP server.

The purpose of this routine is to determine which units are available through the MSCP disk server in the controller. This is accomplished by issuing GET UNIT STATUS commands with the "next unit" flag set until all available units have been reported by the server. The operation begins with unit 1 and works its way "uphill" until it wraps around to unit 0.

As each unit is reported, routine DUTU$NEW_UNIT is called to modify the I/O database to account for the reported unit. DUTU$NEW_UNIT may create new UCBs (and possibly DDBs), and it may setup secondary path linkages for existing UCBs.

There is a special limited form of mount verification performed for the system disk. If the system disk is to be a member of a controller based shadow set, then the associated shadow set virtual unit is created at this time.

2.4.3.1 Polling Loop

- First, DUTU$POLL_FOR_UNITS sets the POLLING flag in the CDDB$W_STATUS field to indicate that polling is in progress for this CDDB.
- Next, it allocates a RSPID and a message buffer in which to build a GET UNIT STATUS command.
- Looping, it issues GET UNIT STATUS commands with the MD_NXUNT modifier set. Starting with unit number 1, DUTU$POLL_FOR_UNITS works "uphill". For each end message received, it does the following:
  - If the ORIGUCB field of the CDDB has been zeroed or if the Unit number/*Server Local Unit Number* (SLUN) returned in the end message does not match the unit in the ORIGUCB, then this is not the boot device.

    In this case, if the end message STATUS field is any of SUCC, AVLBL, or DRIVE, or if the end message Status field is OFFLN with either the NOVOL or EXUSE subcodes set, then the following steps are taken:
    - Load Balancing information is copied from the end message to the CDDB.
    - DUTU$NEW_UNIT is called to modify the I/O database for a possible newly discovered disk unit or a second path to an already known unit.
    - If DUTU$NEW_UNIT either creates a new UCB or reports an already existing one, and if the unit is online (end message STATUS = SUCC), then the UNT_FLGS field of the end message is copied to the UNIT_FLAGS field in the UCB.

    If the STATUS of the end message is not one of these four, then the message (and hence also the unit) is merely ignored.

— If the UNIT/SLUN field in the end message matches the the ORIGUCB field in the CDDB, then the unit is the boot device and routine DO_ORIG_UCB is invoked to perform "special handling":

- o The media and MSCP unit numbers are copied from the end message into the MEDIA_ID and MSCPUNIT fields of the UCB.
- o If the Restart Parameter Block (RBP BootR3) indicates that the system disk is part of a controller based shadow set, then routine DU$SYSTEM_SHADOW_SET (in module DUMNTVER) is invoked for the first time to create a shadow set virtual unit for the system disk:
  - * Creates a UCB to serve as the the system disk shadow set virtual unit UCB by calling DUTU$NEW_UNIT
  - * The system images WCB's UCB pointers are changed to point to the new system disk shadow set virtual unit UCB.
  - * Updates the logicals SYS$DISK and SYS$SYSDEVICE to reflect shadow set usage.
  - * Changes location EXE$GL_SYSUCB to point to the new system disk shadow set virtual unit UCB.
  - * Issues an internal IO$_CRESHAD IRP to effect shadow set creation.
- o An IO$_PACKACK function (ONLINE, GET UNIT STATUS) is issued to the disk as a limited form of mount verification.

NOTE

If the system disk is a member of a shadow set, then the IO$_PACKACK is issued to the shadow set virtual unit.

- o If no IO$_PACKACK error occurs, then the LCL_VALID flag (local valid flag for system device) is set in the STS field of the UCB.

NOTE

In the event that an error occurs, the code loops reissuing the IO$_ PACKACK until it succeeds.

- o If the Restart Parameter Block (RPB) indicates that the system disk is part of a shadow set, then routine DU$SYSTEM_SHADOW_SET is invoked for a second time to copy disk geometry information from the system disk virtual unit UCB to the member UCB used by DU$SYSTEM_SHADOW_SET the first time it was called to create the virtual unit UCB:
  - * Device dependent information (DEVDEPEND).
  - * Total user visible blocks (MAXBLOCK).
  - * LBNs per track (LBNPTRK).
  - * Tracks per group (TRKPGRP).
  - * Groups per cylinder (GRPPCYL).
- o ORIGUCB field in CDDB is cleared.

- If the end message is not for unit 0, then the RSPID and message buffer are recycled and the code branches back up to issue the next GET UNIT STATUS.

2.4.3.2 Polling for Units Complete

After unit numbers in the end messages wrap around to 0 and the unit 0 end message is processed:

- The RSPID is released and the message buffer is deallocated.
- The POLLING flag in the CDDB is cleared.

2.4.4 DUTU$NEW_UNIT

DUTU$NEW_UNIT is called to perform "new unit processing" each time any one of three events occurs:

- When the class driver's controller initialization routine DU$CONTROLLER_INIT establishes a connection with the MSCP disk server in the controller, it calls DUTU$POLL_FOR_UNITS to determine what disks the controller is serving to the cluster. This is done by a series of GET UNIT STATUS commands, each with its "next unit" flag set. For each corresponding end message received from the controller indicating the existence of another disk, DUTU$NEW_UNIT is called.
- After controller initialization, the receipt of a UNIT AVAILABLE ATTENTION message from the MSCP disk server in a controller indicates that a unit has asynchronously become available via that controller to this class driver. DUTU$NEW_UNIT is called upon receipt of each such message.
- During the creation of a shadow set, routine DU$CRESHAD_FDT (called during the *IO$_CRESHAD function*) invokes DUTU$NEW_UNIT to create a UCB to represent the shadow set virtual unit.

NOTE

DU$CRESHAD_FDT constructs a skeleton message containing data equivalent to what would have been received in the above messages. This constructed message is then passed to DUTU$NEW_UNIT as if it had been received from a controller.

DUTU$NEW_UNIT scans the chain of UCBs linked to the CDDB associated with the controller looking for a UCB corresponding to the unit described in the message:

- If a matching UCB is found, then the unit is already known and DUTU$NEW_UNIT does nothing more than report to its caller the UCB's address.
- If a matching UCB is not found, then CDDBs corresponding to other controllers are considered. If a matching UCB is found linked to some other CDDB, then appropriate dual pathing linkages are established by a call to routine *DUTU$SETUP_DUAL_PATH*.

- If no matching UCB is found, then a new UCB is created.

NOTE

Secondary pathing linkages are not established for shadow set virtual unit UCBs.

2.4.4.1 Determines if Unit Already Seen on Controller

Searches down chain of UCBs for this controller looking for a UCB which matches UCB described in the message passed to DUTU$NEW_UNIT (Call to routine *DUTU$LOOKUP_UCB* .

- If the SHADOW flag (bit 15) is set in the UNIT field of the message (indicating that the unit is actually a shadow set virtual unit) but the class driver does not support volume shadowing, then the message is merely ignored and an SS$_IVDEVNAM status is returned.
- DUTU$NEW_UNIT calls DUTU$LOOKUP_UCB to search down the chain of UCBs linked to this CDDB trying to find an already existing UCB for this unit. In essence, it is checking whether or not the unit reported in the message is already known to be on the controller associated with this CDDB.
  - — CDDB$L_UCBCHAIN is the list head. Linkage is via the UCB$L_CDDB_LINK field in each UCB.
  - — To have a match, the following conditions must be satisfied:
    - o The MSCPUNIT field in the UCB and the UNIT field in the message must be identical.
    - o The D0 and D1 media id fields must match
    - o If the device's Server Local Unit Number (SLUN) bit is set indicating a served unit, then the device type (DJ, DU, ...) in the UCB and message must also be the same. If the SLUN bit is clear, then we assume this is the same unit without the device type check.

2.4.4.2 Unit Already Seen On This Controller

If DUTU$LOOKUP_UCB does find a matching UCB linked to this CDDB, then the unit is already known to be on the controller associated with this CDDB. So DUTU$NEW_UNIT merely performs some minor bookkeeping tasks and returns to its caller without doing any further work for this unit.

2.4.4.3 Unit Not Already Seen On This Controller

If DUTU$LOOKUP_UCB does not find a matching UCB linked to this CDDB, then this is the first time the unit has been seen on the controller corresponding to this CDDB. However, it may have already seen this unit on some other controller. If so, what has now been found is actually a secondary path to the same unit.

- Thus, DUTU$SETUP_DUAL_PATH is called to investigate the "secondary path possibility" and set up appropriate secondary path linkages if a secondary path has just been discovered.
  - — Searches all other CDDBs (i.e. CDDBs corresponding to other controllers) for a combination of a CDDB and UCB which satisfies three conditions:
    - o The other CDDB is in the same nonzero allocation class as the CDDB passed to this routine.

      NOTE

      This necessitates two controllers supporting a dual pathed disk to be in same allocation class.

    - o The UCB linked to the other CDDB has the same device type (DJ, DU, ...) as the UCB in the message.
    - o The MSCPUNIT field of the UCB matches the unit number of the unit reported in the message.

    A CDDB and UCB combination satisfying all three of these conditions represents an already established primary path.
  - — If a matching UCB and CDDB combination is found but the UCB represents a shadow set virtual unit (i.e. UCB has SHAD flag set), then secondary path linkages are not established here and DUTU$SETUP_DUAL_PATH does no further work for this unit.

    However, DUTU$SETUP_DUAL_PATH does report to its caller that it has found a matching UCB in this case. Thus it will "appear" as if it had set up dual pathing, even though it hasn't.
  - — If matching UCB and CDDB are found and the UCB does not represent a shadow set virtual unit, then UCB secondary path linkages in the I/O database are established. If a DDB corresponding to the device type (DJ, DU, ...) is not already linked to the SB corresponding to the secondary path controller, then one is created and linked to the SB at this time.

    NOTE

    If secondary path linkages do not already exist for the matching UCB, then the newly discovered path will become the secondary path. However, if secondary path linkages already exist for the UCB, the newly discovered path replaces the old secondary path if and only if the new path is not for an MSCP emulator or if this was a *Load Balance Message*.

    In this case, again DUTU$SETUP_DUAL_PATH reports to its caller that it has found a matching UCB, thus indicating that it has set up dual pathing for the unit.

- If DUTU$SETUP_DUAL_PATH reports finding the UCB (and thus dual pathed it, or at least "appears" to have done so), then DUTU$NEW_UNIT merely performs some minor bookkeeping tasks and returns to its caller without doing further work for this unit.

  If DUTU$SETUP_DUAL_PATH reports not finding a UCB with same unit number linked to some other CDDB in same allocation class as this CDDB, then DUTU$NEW_UNIT:

  — Calls *IOC$COPY_UCB* to create a new UCB from the "template" UCB pre-allocated in the class driver itself.

  — Invokes *LINK_NEW_UCB* to link the new UCB into the database. (A new DDB will also be created if necessary.)

2.4.5 DUTU$DODAP

Determine Access Paths processing is invoked to find as yet unknown secondary paths to dual-ported disks. It is invoked periodically by DUDRIVER's CRB-based timeout mechanism in routine DU$TMR.

2.4.5.1 Preparations for Performing DAP Processing

DUTU$DODAP begins by verifying that DAP processing should actually be performed at this time.

- It examines the DAPBSY flag in the CDDB$W_STATUS field.
  — If this flag is already set, then previous DAP processing for this CDDB has not yet completed. So this fork thread merely terminates itself.
  — If the DAPBSY flag is not already set, then it is set here and this fork thread continues.
- If this is a VMS MSCPserver (emulator) then exit
- The DAPCOUNT field in the CDDB is decremented.
  — If the DAPCOUNT field is still greater than or equal to 0, the DAPBSY flag is cleared and this fork thread terminates itself without doing further work.
  — If decrementing the DAPCOUNT field has now made it negative, then it is reset to the value of parameter DAP_COUNT and this fork thread is allowed to proceed.
- It allocates a RSPID and associated RDT entry for DAP processing.

2.4.5.2 Issues DAP Commands to Controller

DUTU$DODAP now loops through the chain of UCBs linked to the CDDB. It issues a DAP command to the controller associated with the CDDB for each "qualified" unit on that controller.

- For each UCB linked to the CDDB:
  - — Verifies that the unit is qualified for DAP processing. To be qualified, two conditions must both be met:
    - o VALID flag is currently set in the UCB$L_STS field (indicating unit is currently MSCP online).
    - o UCB does not represent a shadow set virtual unit (i.e. MSCP$V_SHADOW bit is not set in the MSCPUNIT field of the UCB).
  - — If the unit is not qualified, this routine merely skips it and branches back to consider the next UCB in the chain.
  - — If the unit is qualified for DAP processing:
    - o A message buffer is allocated.
    - o A DAP command is built for this unit in the message buffer.
    - o The message buffer is passed to the SCS and PPD layers for transmission to the Server.

NOTE

This fork thread is suspended here until the end message corresponding to the DAP command is received from the controller. Then the fork thread resumes by deallocating the buffer containing the end message, recycling the RSPID, and branching back for the next UCB linked to CDDB.

- After the chain of UCBs linked to the CDDB is exhausted:
  - — The RSPID is released.
  - — The DAPBSY flag is cleared.
  - — This fork thread is terminated.

2.4.6 ATTN_MSG

ATTN_MSG in the class driver's *Input Dispatching Routine* (DU$IDR) dispatches to one of three specific attention message handler routines based on the type of attention message if the message is valid. If the type is not valid, it branches to a routine to reset the controller.

- Dispatch is made to a specific routine based on the type of attention message if it is one of the valid types:

Type of Attn Msg	DUDRIVER Routine
Unit Available	UNIT_AVAILABLE
Duplicate Unit	DUPLICATE_UNIT_ATTN
Access Paths	ACCESS_PATH_ATTN

NOTE

Just prior to dispatching, ATTN_MSG pushes onto the stack the address *EXIT_ATTN_MSG* to which specific routines will return for deallocation of the buffer containing the attention message. If the attention message is not valid, the code to handle the invalid case merely pops this address from the stack.

- If the attention message is not valid (i.e. is not one of the above types):
  - — The invalid attention message is logged with error code EMB$C_INVATT.
  - — The buffer containing invalid attention message is deallocated.
  - — A branch is taken to routine DU$RE_SYNCH to reset the controller on the presumption that the controller must be "very ill" if it issued an invalid attention message.

2.4.6.1 Unit Available Attention Message

This routine processes unit available attention messages. It calls a routine which either adds the unit to the I/O database if is unknown, or alters the I/O database to reflect a secondary path to a known unit.

- DUTU$NEW_UNIT is called to alter the I/O database to reflect a new unit or a secondary path to an already known unit, as appropriate.
- If DUTU$NEW_UNIT returns the address of a newly created UCB or the address of a UCB for which a secondary path has just been established, the AVL (unit available) flag in the UCB is set.

NOTE

If the I/O database already reflects a known unit as being dual pathed, DUTU$NEW_UNIT may ignore this path. In such a case, it will not return the address of any UCB. See the section detailing routine DUTU$NEW_UNIT.

2.4.6.2 Duplicate Unit Attention Message

MSCP$DISK in a controller is notifying this host that two or more units of the same device class on that controller have the same unit number. Thus, the operator is notified of this condition.

- Message is sent to operator.
- EMB$C_DUPUN error is logged.

2.4.6.3 Access Paths Attention Message

This routine simply calls DUTU$SETUP_DUAL_PATH to alter the I/O database to properly reflect a secondary path to an already known unit.

NOTE

There once was code here to log an EMB$C_ACPTH error in the event that DUTU$SETUP_DUAL_PATH could not find a UCB corresponding to the unit which was supposed to already be known. In V4.6-V5.5 of VMS, however, this code is effectively NOPed and the access paths attention message is merely ignored. This may be changed in later releases.

2.4.7 Routines in the CONFIGURE Process

The CONFIGURE process consists of two major routines. The first, *BOO$CONFIGURE*, requests the SCS process poller to locate MSCP servers (disk and tape) on other nodes. The second, *FOUND_PROC*, handles messages received from the SCS process poller via a mailbox when a sought after server is found.

2.4.7.1 Polling for MSCP Servers on Other Nodes

BOO$CONFIGURE requests the SCS process poller to poll for MSCP$DISK on other nodes by making a call to SCS$POLL_MBX.

- Using the $CREMBX system service, BOO$CONFIGURE creates a permanent mailbox to communicate with the SCS process poller.
  - — Assigns an I/O channel *Channel Control Block* (CCB).
  - — Allocates and initializes the mailbox UCB.
    - o Flags set in UCB: MBX, PRMMBX, ONLINE
    - o Fields initialized in UCB: OWNUIC, UNIT, DEVBUFSIZ

— Address of UCB stored in CCB.
— UCB linked into mailbox controller's device list via UCB$L_LINK field.
— I/O channel number returned to the CONFIGURE process by the $CREMBX system service.

NOTE

The mailbox is effectively treated as a virtual device.

- Next, BOO$CONFIGURE calls REQ_POLL to request SCS process polling. REQ_POLL loops, calling SCS$POLL_MBX for each SYSAP name for which polling is to be requested. The MSCP disk server name to poll for is MSCP$DISK.
  — SCS$POLL_MBX is passed the I/O channel number of the mailbox which the SCS process poller will use to notify CONFIGURE when a sought after SYSAP is found.
  — SCS$POLL_MBX returns the address of the *SCS Process Polling Block* (SPPB) it creates for each SYSAP to be polled for, and this SPPB address is saved by CONFIGURE in a process information block.

NOTE

Each process information block contains a SYSAP name, associated device name (e.g. DU) and driver name (e.g. DUDRIVER), and SPPB address field. The addresses of these process information blocks are kept in list at PROC_INFO.

- Using the $QIO system service (FUNC = IO$_SETMODE!IO$M_WRTATTN), BOO$CONFIGURE creates a *"write attention" AST* by which it will be notified anytime the mailbox receives a message.
  — Creates ACB (AST address = FOUND_PROC) and queues it to UCB$L_MB_W_AST.
  — MB_W_AST and ASTQFL are both overlays of FPC field in the UCB.
- CONFIGURE then hibernates.

2.4.7.2 CONFIGURE Notified of Discovery of MSCP$DISK

AST routine FOUND_PROC is invoked when the mailbox is written by the SCS process poller. FOUND_PROC requeues the "write attention" AST, reads and processes messages from the mailbox until the mailbox empty, and then resumes hibernating.

- Using the $QIO system service, it requeues the "write attention" AST to the mailbox UCB.
- Loops, processing all messages in the mailbox.

  Using the $QIO system service (FUNC = IO$_READVBLK!IO$M_NOW), FOUND_PROC reads a message from the mailbox into a local buffer, MSGBUF, and then calls PROCESS_MSG to process the message as follows:
  — Calls routine BLDNAME to construct a cluster device name of the form: *Node_name*$DUA)

— Calls routine BOO$CONNECT to build the class driver database, load the class driver if it has not already been loaded, and call the class driver controller initialization routine DU_CONTROLLER_INIT.

- To build the class driver database, BOO$CONNECT performs various minor supporting tasks and then calls IOGEN$LOADER to do the real work. IOGEN$LOADER performs the following tasks:
  — If DUDRIVER has not yet been loaded, it is loaded here.
  — Allocates a DDB corresponding to *Node_name*$DUA (actually just DUA).
  — Initializes NAME, DRVNAME, SB, and ALLOCLS field in the DDB.

NOTE

ALLOCLS field initialized at this time to local node's allocation class. Changed to correct value later by controller initialization in DUDRIVER.

  — Allocates and initializes CRB and IDB.
  — Allocates a UCB and initializes the UCB$W_UNIT field to 0 at this time.
  — Address of UCB placed in list of UCB addresses starting at offset UCBLST in IDB.
  — UCB queued to DDB.
  — Initializes UCB and ORB based on DUDRIVER prologue table.
  — Calls controller initialization routine, DU_CONTROLLER_INIT, in DUDRIVER.

Chapter 3

$QIO System Service and DUDRIVER

3.1 Introduction

This chapter presents the flow of a typical file read and write request for devices handled by the Disk Class Driver (DUDRIVER). DUDRIVER's immediate involvement in these $QIO operations will be covered, as well as the "pre-processing" and "post-processing" surrounding this involvement.

To properly set the stage for doing file reads and writes, a user process "assigns" an I/O channel to the disk on which the file resides. This act establishes a logical path to that device.

The user process then "opens" the file through the use of an ACP QIO function. This function will return mapping information about the file. The mapping information describes the location on the disk of the blocks associated with the particular file

Certain major steps performed by these tasks are also presented in this chapter since they are essential to the topic.

3.2 Assigning an I/O Channel to a Disk

3.2.1 Assign System Service

Before a process performs input or output with a device, a logical software path must first be established between the process and the device. This logical software path is called an I/O channel. In essence, the collection of information maintained by the operating system which describes the device and how to access it must be looked up and made available to the process.

However, it would be extremely inefficient to perform this lookup as part of every I/O operation. Consequently, VMS provides the system service *SYS$ASSIGN* to perform this task once for a process. In so doing, SYS$ASSIGN returns to the process what amounts to a "pointer" to the information. This "pointer" is known as an *I/O channel number.*

The collection of information referred to in this conceptual explanation is the *Unit Control Block* (UCB) associated with a device. As was pointed out in the chapter on DUDRIVER's I/O database, the UCB identifies the media, unit number, characteristics, and status of a disk unit.

The UCB also maintains the address of the CDDB containing the queue of active MSCP commands for this unit, and class driver specific information about the controller.

The address of the DDB which provides the name of the driver for the disk is found in the UCB as well. The DDB also points to a *Driver Dispatch Table* (DDT) containing the addresses of entry points in the driver for such tasks as starting an I/O operation, canceling an I/O operation, and performing unit initialization.

3.2.2 Channel Control Blocks

One of the tasks of the SYS$ASSIGN system service is to lookup the UCB for the device and store its address in a data structure known as a *Channel Control Block* (CCB). Each process has its own collection of CCBs. They are kept in an array in the process's P1 space, the base address of which is in location *CTL$GL_CCBBASE*. The channel control blocks are allocated toward decreasing memory addresses.

NOTE

CTL$GL_CCBBASE is not itself the base address of this array, but rather is a P1 address location containing the base address of the array.

3.2.2.1 Maximum Channel Limit

Sysgen parameter *CHANNELCNT* determines the number of CCBs allocated in this array when a process is created. The value of this parameter is stored in location *SGN$GW_PCHANCNT*.

3.2.2.2 Channel Number

The I/O channel number returned to the process by the SYS$ASSIGN service is an offset relative to the "base address" of this array, identifying the particular CCB in which SYS$ASSIGN stored the UCB address.

CCBs are stored at negative offsets relative to this base address. Thus, for example, if the I/O channel number returned by SYS$ASSIGN is 100, then the address of the associated CCB is computed as follows:

<Content of CTL$GL_CCBBASE> - 100

The CCB corresponding to I/O channel number 0 is never given out to a process by SYS$ASSIGN. This CCB is reserved by the operating system for error detection.

There are three fields in the CCB of particular interest to the general flow of a disk read or write:

- *CCB$L_UCB*

  This is where SYS$ASSIGN stores the address of the UCB describing the disk for which an I/O channel has been assigned.

- *CCB$L_WIND*

  The address of a *Window Control Block* (WCB) providing mapping information used to determine where on the disk each of the blocks of the file are located.

  This field is not filled in by the SYS$ASSIGN, but rather when the file is opened. It is discussed later in this chapter.

- *CCB$B_AMOD*

  This field contains the mode plus 1 of the process at the time the I/O channel was assigned, or 0 if the CCB is not in use.

3.2.2.3 Numerical Representation of Access Mode

Value	Access Mode
0	Kernel
1	Executive
2	Supervisor
3	User

SYS$QIO system service code can simultaneously verify that a CCB is in use and that the process is currently allowed to access the I/O channel in one operation. If the access mode of the process at the time it attempts a $QIO operation is numerically less than the content of the CCB$B_AMOD field, then both conditions are true. Otherwise, either the the CCB corresponding to the channel is not in use, or the process is not in a sufficiently high access mode.

For example, assume that the process was in executive mode when the channel was assigned. Then the AMOD field will contain a 2 (1 higher than the mode when the channel was assigned). If the process is now in kernel (0) or executive (1) mode, then its current mode is numerically less than the content of the AMOD field; so the CCB is in use and the channel may be accessed. However, it the process is in supervisor (2) or user (3) mode, then the process's current access mode is greater than or equal to the content of the AMOD field; so the process may not access the channel.

This test is one of those applied toward validating parameters supplied by a process when it attempts a $QIO to read or write a disk file.

A related P1 space location of general interest is *CTL$GW_CHINDX*. It contains the highest I/O channel number (CCB index) assigned during the life of the process.

3.2.3 Volume Set Considerations

A volume set is a collection of volumes that are treated as if together they constitute a large single volume. Created through "binding" two or more volumes together at mount time, they handle situations such as where a database is too large to fit on a single volume, or where it is desirable to have a "very large" public file space. In a volume set, one volume takes on special importance by virtue of having the *Master File Directory* (MFD) for the entire volume set; this volume is called the *"root volume"*.

When a process assigns an I/O channel to a volume set, it will be the UCB for the root volume whose address gets stored in the CCB$L_UCB field. Also, the logical name for a volume set is associated only with the root volume.

3.2.4 Overview of Steps Taken by SYS$ASSIGN

The following is a brief summary of the steps taken by routine *EXE$ASSIGN* to implement the assignment of an I/O channel to a disk.

- Routine *IOC$FFCHAN* is called to search the array of CCBs in the process's P1 space for an unused CCB (i.e. one whose CCB$B_AMOD field contains a 0).
- Routine *IOC$SEARCH* is called to search the I/O database for the UCB associated with the device.

  If a logical name is specified, it is translated to a device name. The search then commences in routine *IOC$SEARCHINT*. The steps taken in the search depend on whether the device name involves the node name of the controller, or an allocation class.

  — Assume that the node name of the controller is used, and not an allocation class. As an example, consider the device name HSC001$DUA2.
    - o The SB for HSC001 is found by scanning the queue of SBs whose head is SCS$GQ_CONFIG.
    - o The list of DDBs attached to this SB (list head at offset SB$L_DDB) is searched for the generic controller type DUA.
    - o The list of UCBs attached to this DDB (list head at offset DDB$L_UCB) is searched for the UCB corresponding to unit number 2.

      The secondary path linkage via the offset DDB$L_2P_UCB is not searched.

  — If the allocation class format is used, for example 255DUA2, then these are the steps used in the search:
    - o The first SB in the queue of SBs is selected for consideration.
    - o The DDB list attached to the SB is searched for the generic controller type DUA and allocation class 255. If a matching DDB is found, then the UCB lists (first the primary, and then the secondary) are searched for a UCB corresponding to unit 2.
    - o If the DDB and UCB are not found, the preceding step is repeated for each succeeding SB until either they are found or the queue of SBs is exhausted.
- The address of the UCB and the process's "access mode + 1" are written into the CCB.

- The index of the CCB is returned to the process to serve as the I/O channel number.

3.3 Opening a File

After an I/O channel is assigned for a disk, the process opens a file on that channel using a $QIO. In doing this $QIO, the process specifies a major function of *IO$_ACCESS* to request a directory lookup, and a function modifier of *IO$M_ACCESS* to actually open the file.

By means of other parameters, the process may also specify whether it desires both read and write access, or just read access, whether to allow others to read or write the file, if write checking should be enabled, etc.

Opening a file involves a great many file system operations. Most of these operations are unimportant to understanding the flow of file reads and writes. They belong to the domain of file system internals, and are thus not presented in the context of this book. However, the "mapping" information made available by opening a file is essential.

3.3.1 Window Control Blocks and Mapping a File

As a result of the $QIO to open a file on an already assigned I/O channel, a data structure known as a *Window Control Block* (WCB) is allocated. The address of the WCB is stored in the CCB$L_WIND field of the CCB associated with the channel. The purpose of this data structure is to assist in determining where on the disk the blocks of the file are located. This then brings up the terminology of *Virtual Blocks*, *Logical Blocks*, and *Physical Blocks*.

3.3.1.1 Virtual Blocks

To the user process, a file appears as a contiguous stream of "virtual blocks", that is, the blocks are numbered "uphill" relative to the beginning of the file, starting with virtual block number 1. The placement of a file's blocks may not be physically contiguous on the disk; this is referred to as *File Fragmentation*.

A virtual block number does not directly indicate the actual location of the block on the disk. A translation mechanism is required to convert a virtual block number to an actual disk address; and this mechanism is dependent on the type of disk used. These issues are transparent to the process. It continues to have the "illusion" of the file being an unbroken stream of blocks starting with a block numbered 1.

3.3.1.2 Logical Blocks

On DSA disks, actual disk addresses are called "logical block numbers". To a VAX host, a DSA controller presents its disk as a consecutive stream of blocks numbered from 0 through N-1, where N is the number of logical blocks on the disk. The blocks are called "logical" because the controller, and not the host, manages the actual geometry of the disk.

The host operating system does not have to deal with the traditional concepts of "cylinder", "track", and "sector" that are associated with the older MASSBUS and UNIBUS disks. For normal disk reads and writes, there are basically only two geometry and addressing issues the host is concerned with:

- The total number of logical blocks available to the host on the particular disk.
- Translating a *virtual block number* (VBN) relative to the beginning of a file into a *logical block number* (LBN) relative to the beginning of a disk.

3.3.1.3 Bad Block Replacement

Another feature of DSA disks is that *Bad Block Replacement* (BBR) is handled by the controller if that controller is "sufficiently intelligent"; otherwise, BBR is handled as a cooperative effort between the disk class driver and the controller.

BBR on DSA disks is accomplished by maintaining a pool of *Replacement Blocks* used to replace host area logical blocks containing media defects leading to hard errors or large numbers of correctable errors. When DUDRIVER references a logical block number which has been "replaced", the reference is "revectored" to a *Replacement Block Number* (RBN) by the controller without further intervention by DUDRIVER.

What about disks being served on a remote VAX? Since the VMS based MSCP server on the remote VAX is emulating an HSC, other VAXes send MSCP read and write commands to it just as they would to an HSC. Consequently these commands contain logical block numbers, regardless of the type of disk being served.

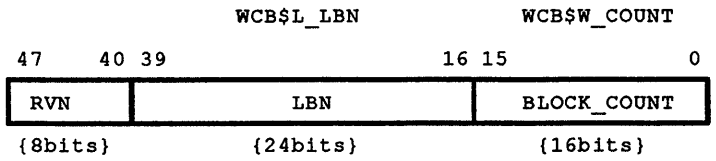
If that disk is on a DSA controller, the LBN is passed unaltered to the remote node's DUDRIVER by the remote MSCP server. If it is a MASSBUS or UNIBUS disk, the logical block number is first converted by the MSCP server to a traditional "physical block" number involving a cylinder, track, and sector. Then it is passed to the driver for that particular device.

3.3.1.4 Window Control Blocks

Given this explanation of VBNs and LBNs, the role of the *Window Control Block* can now be summarized by saying that it provides the mechanism used by the $QIO system service for mapping (i.e. converting) VBNs to LBNs[1]. In essence, it is a cache of VBN-to-LBN translations contained in 48-bit entries, each with the format as in Figure 3–1:

[1] The 24 bit LBN field in the WCB limits the maximum disk size currently to approximately 16.7 million blocks

Figure 3–1: Window Control Block Fields for VBN to LBN Translation



CXN-0003-12

BLOCK_COUNT — Number of consecutive logical blocks represented by this entry.

LBN — Starting logical block number.
This is the LBN of the first block in the set of consecutive logical blocks represented by this entry.

RVN — Relative volume number.
An RVN field of 0 indicates that the disk involved is not part of a *volume set*. If this field is nonzero, then it is the relative volume number of a unit within the volume set. The volumes within a volume set are numbered in ascending order, starting with the root volume being relative volume 1.

Each entry represents a set of consecutive logical blocks. The entire collection of valid entries in the WCB represents a set of consecutive virtual blocks, starting with the VBN stored at offset *WCB$L_STVBN*. Consider as an example the case of where the WCB$L_STVBN field contains a 1 and the first three entries are as shown in Figure 3–2.

Figure 3–2: Window Control Block Mapping

	wcb$l_lbn	wcb$w_count
0	1001	4
0	2001	8
0	5001	2

CXN-0003-13

In this situation, the file is fragmented. The first entry in this WCB maps the file's virtual blocks 1, 2, 3, and 4 to disk logical blocks 1001, 1002, 1003, 1004, respectively. The second entry takes up with the file's virtual blocks where the first entry left off. The second entry maps the next 8 virtual blocks, namely 5 through 12, to disk logical blocks 2001 through 2008 respectively. And finally the third entry maps only 2 virtual blocks, 13 and 14, to logical blocks 5001 and 5002, respectively.

The actual number of currently valid entries in a WCB is stored at offset *WCB$W_NMAP*. The maximum number of entries, however, depends on the values of three parameters. The default number of entries is the value of the sysgen parameter *ACP_WINDOW*. This value is stored in location *ACP$GB_WINDOW*. This default may be overridden for a disk by using the "/WINDOWS" qualifier when it is initialized. Both of these may optionally be overridden for a particular file when it is opened by the $QIO system service.

3.3.2 Mapping Situations Requiring Special Handling

There are special cases whereby multiple WCBs are chained together to form a *cathedral window* for mapping an entire file. The average user generally uses only one WCB. Furthermore, that WCB usually has only a moderate number of mapping entries.

The default number of entries in a WCB is set by the Sysgen Parameter ACP_WINDOW and equals seven. Since the block count field of each WCB entry is limited to 16 bits in size, file extents exceeding 65535 contiguous blocks require multiple WCB entries. When the $QIO system service code goes to map a VBN to an LBN, it can run into three additional situations which require a bit of special handling:

3.3.2.1 Window Turns

The VBN to be mapped is out of the range of VBNs handled by the current entries in the Window Control Block.

This is resolved by invoking the XQP to refill the WCB from the file header with a new set of entries. The first entry will map the desired VBN. This event is known as a *window turn*.

3.3.2.2 Bound Volume Sets

The UCB whose address is stored in the CCB is for the root volume of a volume set, but the entry mapping the VBN indicates the corresponding LBN is on one of the other volumes in the set.

This is actually handled automatically by the routine, *IOC$MAPVBLK*, which maps VBNs to LBNs. If it finds the RVN field of the WCB entry is nonzero, it merely uses the RVN as an index into a list of UCB addresses stored in another data structure called the *Relative Volume Table* (RVT).

The UCB corresponding to the proper unit in the volume set is located using the Relative Volume Number as an index into the list of UCB addresses that are stored starting at location *RVT$L_UCB*.

From this point on, the processing of a $QIO uses this new UCB address. This does not alter the content of the CCB$L_UCB field; this longword continues to contain the address of the UCB corresponding to the root volume of the volume set.

NOTE

The address of the RVT is found at offset WCB$L_RVT within the WCB.

3.3.2.3 File Fragmentation

Quite frequently, IOC$MAPVBLK is called upon to map a set of consecutive VBNs rather than just one. Due to file fragmentation, this may involve more than one entry in the WCB. Each entry taken by itself represents a set of consecutive LBNs, but there are gaps between the sets of LBNs represented by a pair of entries. Alternatively, the current WCB may map the first few VBNs, but mapping the remainder would require a window turn.

Starting with the first VBN, routine IOC$MAPVBLK will map as much of the $QIO request as it can to a set of consecutive logical blocks called a *segment*. It will return to its caller the "starting LBN" (i.e. the LBN corresponding to the first in the set of VBNs it mapped). The number of bytes which were not mapped due to being unable to map the entire set of VBNs to consecutive LBNs will also be returned.

If a $QIO request cannot be mapped to a single consecutive set of LBNs without a window turn, DUDRIVER will be asked to handle the request in portions called *transfer segments*. As each segment completes, the next segment will be mapped and passed to DUDRIVER until the request is completely satisfied. This activity, however, is transparent to the process; so the process does not have to issue repeated $QIOs if file fragmentation exists.

3.4 Driver Data Structures, the IRP, DDT and FDT

Three more data structures play an essential role in the flow of a $QIO: the *I/O Request Packet* (IRP), the *Driver Dispatch Table* (DDT), and the *Function Decision Table* (FDT).

3.4.1 I/O Request Packet

When a process queues an I/O request for some device, the $QIO system service code allocates and initializes an I/O Request Packet (IRP). The purpose of this data structure is to describe the request to the particular driver which will handle it. Here are some of the typical items of information found in an IRP:

- Process identification of the process which issued the request represented by the IRP.
- Address of the process's quadword *I/O Status Block* (IOSB) into which final completion status is to be written.
- Function code identifying the type of I/O operation (read, write, etc.).
- I/O channel number for the request (representing the Channel Control Block).
- Address of the UCB corresponding to the device for which the I/O operation is to be performed.
- Address of, or "pointer" to the buffer for holding the data to be read from or written to the device.
- Number of bytes to be transferred.

These data items are fairly generic in the sense that they apply to most any device driver. In fact, most of the fields of an IRP are driver *independent*. They are initialized on the basis of parameter values supplied by the process making the I/O request. Then the IRP is passed to the driver which will actually handle the request.

In describing an I/O request, it is very common for a driver to supplement the information in a generic IRP with additional data which is specific to the type of device involved. Thus an IRP often has a driver specific extension, the address of which is stored in a longword near the end of the IRP itself. Data contained in the extension is placed there by the driver. In the case of DUDRIVER, this extension is a *Class Driver Request Packet* (CDRP).

The information contained in the IRP is common to most devices. A driver must translate that information into a command format which is meaningful to the controller for the device to which the request is directed. Thus, the disk class driver allocates another buffer, builds an MSCP command in that buffer based on the content of the IRP, and then stores the address of that buffer in the CDRP. This address is a major component of the driver specific information supplementing the content of the IRP.

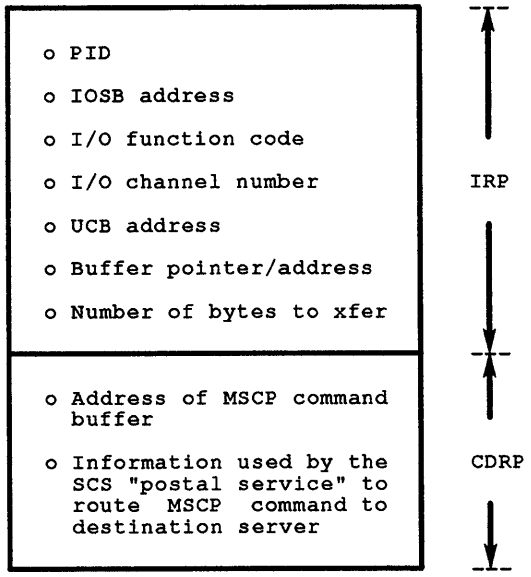
3.4.1.1 Class Driver Request Packet

The CDRP is chosen as DUDRIVER's extension to the IRP rather than some other data structure for the following reason. From the chapter covering SCA concepts, the purpose of a CDRP is to facilitate a SYSAP making a request for service from the SCS layer of software; hence the name "Class Driver Request Packet". The disk class driver is performing that function.

In addition to the address of the MSCP command buffer, the class driver also places in the CDRP information identifying the node (or controller) and server to which the MSCP command is to be sent. It then passes the CDRP to the SCS layer, requesting transmission of the MSCP command. In essence, the MSCP command is the "letter", the CDRP is the "envelope", and the SCS layer of software is the "postal service".

IRPs are allocated using the standard nonpaged pool allocation routine. So that two separate operations are not required for the allocation of IRPs and CDRPs, each IRP pre-allocates the extra space for a CDRP. The fields in an IRP can be treated as negative offsets from the beginning of the CDRP since the first byte of the CDRP immediately follows the last byte of the generic IRP. Thus, when the $QIO system service allocates an IRP, it implicitly allocates a CDRP. For this reason, together they are often referred to as a *IRP/CDRP pair*. Some of the major fields of the IRP/CDRP pair are displayed in Figure 3–3.

Figure 3–3: IRP/CDRP pair organization



It should be noted that IRPs and CDRPs are really distinct data structures, and that the pairing just described is done for purposes of convenience and efficiency. Other SYSAPs also use CDRPs to make requests of SCS. Some SYSAPs such as the Connection Manager have no use for IRPs. They utilize a facility for allocating just CDRPs by themselves.

3.4.2 Driver Dispatch Table

There exists a generic class of operations and corresponding routines common to most device drivers in VMS. For example, device drivers typically have the following routines:

Routine	Application
Start I/O	Location to which IRPs are initially handed by $QIO system service code
Cancel I/O	Invoked to cancel requested I/O operations before they complete
Register Dump	Used to obtain the contents of various registers for diagnostic and error logging purposes
Unit Initialization	Used to setup initial data structures and conditions for the device

By their very nature, such routines are specific to a particular device or class of devices. It is therefore necessary that each driver have a table listing the entry points for its own set of these routines. This table is called a Driver Dispatch Table (DDT).

Normally, the DDT should be defined at the beginning of the driver; and its address is always stored in the UCBs and DDBs associated with that driver.

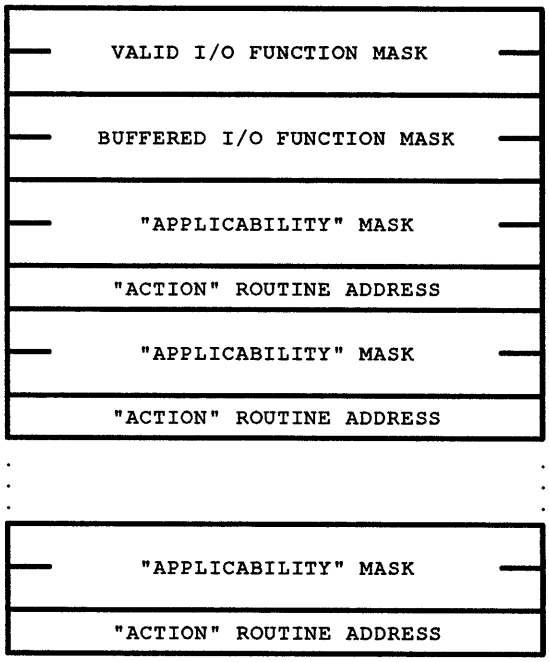
3.4.3 Function Decision Table

Another very important quantity kept in the DDT is the address of another driver specific table called the Function Decision Table (FDT).

A driver's FDT provides a mechanism for validating the I/O function code by verifying that the requested function is valid for the devices handled by the driver. It also contains the addresses of routines to process device and function dependent $QIO parameters and then pass the IRP to the driver.

Figure 3–4 illustrates an FDT, and is the basis for the discussion which follows it.

Figure 3–4: FDT layout



CXN-0003-02

3.4.3.1 Valid I/O Function Mask

The first quadword forms a bit mask called the *Valid I/O Function Mask*. It represents a legal function bit mask of all I/O function codes which are valid for the devices handled by this driver. A function code, being a 6-bit unsigned quantity, has a numerical value in the range of 0 to 63. Thus it is used as an index into the quadword Valid I/O Function Mask to determine whether or not the requested operation is legal. If the bit corresponding to the function code is set, then the operation is legal; if the bit is clear, the operation is illegal. ($QIO system service code uses a BBC instruction to make this determination.)

For disk devices, this is sufficient validation of the function code since they normally do *direct I/O*. The process pages containing the buffer into which data is to be read from disk, or from which data is to be written to disk, are locked in physical memory and mapped to system space. In this way the buffer is always addressable by the driver.

3.4.3.2 Buffered I/O Function Mask

Some devices such as line printers perform *buffered I/O*. Data is transferred from the process's buffer to an intermediate system buffer which is always available to the driver, even when the process is not in physical memory. The second quadword is a *Buffered I/O Function Mask* for validating those I/O functions that are buffered.

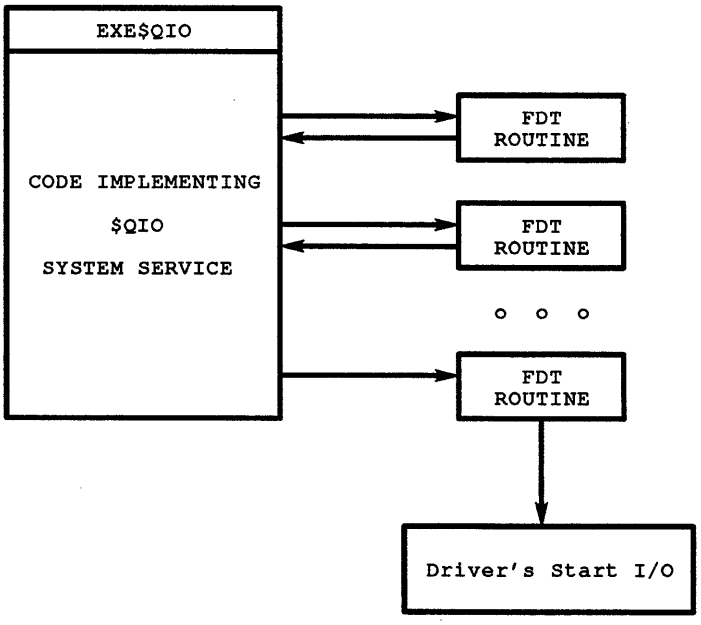
3.4.3.3 Applicability and Routine Entries

After the two validation bit masks, each entry consists of three longwords. The first two longwords collectively form a 64-bit *Applicability Mask*. If the bit in the mask corresponding to the function code is set, then the entry applies to that function code; otherwise, it doesn't.

$QIO system service code loops, scanning these entries. For each entry it finds that applies to the requested function, it calls the "action" routine whose address is in the entry's third longword. These are the routines that have the responsibility for processing the device and function *dependent* $QIO parameters.

The action routine that completes device and function dependent parameter processing has the added responsibility of branching to code which hands off the IRP to the driver's Start I/O routine; it does not return to the loop which called it. Function Decision Table processing is illustrated in Figure 3–5.

Figure 3–5: FDT processing



NOTE

With some drivers and some operations, the last FDT routine may instead branch to code to complete or abort the request.

The preceding discussion of FDT processing is intended to explain the general case. But in the sections which present the flow of a $QIO later in this chapter, it will be seen that there is only one FDT routine for each of the basic disk read and write functions.

3.5 Overview of the Flow of a $QIO

A queued I/O operation to transfer data to or from a disk file begins with a process specifying parameters which define the operation. Then it invokes the $QIO system service to pass these parameters to VMS, and to request that the operation actually be performed.

There are various approaches to setting up system service parameters and calling the system service. If a program is being written in VAX/VMS Assembly Language, then a rather straight forward mechanism is to use various macros defined in the system libraries provided with VMS. One such macro, defined in SYS$LIBRARY:STARLET.MLB, is $QIO_S.

At the point in the program where a file read or write is desired, the programmer uses this macro to specify the parameters to be passed to the routines in VMS which implement the $QIO system service. Keywords are used to specify which parameters are being used. At assembly time, the macro expands into a sequence of instructions which set up an argument list based on the specified parameters, and provide default values for unspecified optional parameters. The macro then generates the instruction for actually calling the $QIO system service.

Here is the generic format for setting up a file read or write $QIO using this macro:

```
$QIO_S    CHAN   = ... ,
          FUNC   = ... ,
          IOSB   = ... ,
          EFN    = ... ,
          ASTADR = ... ,
          ASTPRM = ... ,
          P1     = ... ,
          P2     = ... ,
                . . .
          P6     = ...
```

Parameters passed to the $QIO system service code are listed in Table 3–1:

Table 3–1: QIO System Service Parameters

Parameter	Description
CHAN	Address of the longword containing the I/O channel number assigned to the device on which the file resides. (This I/O channel is also associated with with the file which has already been opened on the channel.)
FUNC	I/O function code which specifies if the requested operation is a read (IO$_ READVBLK) or a write (IO$_WRITEVBLK).
IOSB	Address of the quadword I/O status block into which will be stored a system service completion status code and the number of bytes transferred. (While technically optional, good programming practice dictates that this always be supplied.)
EFN	Event flag that is to be set upon completion of the system service. ([Optional parameter]. Usually considered redundant if a programmer specifies an AST to be used for signaling completion of the request.)
ASTADR	Address of the entry mask for a programmer specified AST service routine which is to be executed upon completion of the system service. ([Optional parameter]. Usually considered redundant if a programmer specifies an event flag to be used for signaling completion of the request.)
ASTPRM	AST parameter. A longword passed to the programmer specified AST service routine, if there is one defined by the ASTADR parameter. ([Optional parameter]. One application of this parameter arises when different $QIOs use a common AST service routine. If each $QIO uses a different value for this parameter, then the AST service routine can easily determine for which particular $QIO's completion it is being executed.)
P1	[Optional parameter], device and function dependent parameter. For a Disk transfer QIO, this contains the buffer address.

Table 3–1 (Cont.): QIO System Service Parameters

Parameter	Description
P2	[Optional parameter], device and function dependent parameter. For a Disk transfer QIO, this contains the size of the transfer in bytes.
P3	[Optional parameter], device and function dependent parameter. For a Disk transfer QIO, this contains the starting Virtual Block number.
P4	[Optional parameter], device and function dependent parameter.
P5	[Optional parameter], device and function dependent parameter.
P6	[Optional parameter], device and function dependent parameter.

The following example shows a $QIO set up specifically to read two consecutive virtual blocks starting at virtual block 3 from a file into a BUFFER . The I/O channel number returned by a previously executed $ASSIGN is stored in the longword at location DEV_CHN, and the address of the quadword I/O status block is IO_STS_BLK. An ACP QIO *IO$_ACCESS* function has also been executed already to identify the file for the transfer.

```
                .  .  .
DEV_CHN:        .WORD    1
IO_STS_BLK:     .QUAD    1
BUFFER:         .BLKB    1024
                .  .  .
                $QIO_S  CHAN = dev_chn,-
                        FUNC = #IO$_READVBLK,-
                        IOSB = io_sts_blk,-
                        EFN  = #1,-
                        P1   = buffer,-
                        P2   = #1024,-
                        P3   = #3
                .  .  .
```

3.5.1 The Process's Point of View

From the process's point of view, the execution of a $QIO occurs in two major phases: queuing the I/O request to the driver, and the driver handling the request. Associated with each phase is a separate condition value returned to the process at the end of that phase.

3.5.1.1 Queuing the Request to the Driver

First, the parameters defining the request are validated. The I/O channel specified by the CHAN parameter must already be assigned and accessible to the process, and a file must currently be open on that channel. The operation requested by the FUNC parameter must be valid for the device associated with the I/O channel, and that device had better be online.

If parameter validation is successful, the $QIO system service code passes the request to the driver which handles the device.

The end of this first phase is signaled by a condition value being returned to the process in register R0 and control being passed back to the instruction following the $QIO request.

This condition value reflects whether or not the request was successfully passed to the driver. In the event of an error, the condition value returned will indicate the reason for the failure. It does not in any way indicate how successful the driver was in handling the request. Such information will be returned to the process later in a second condition value returned in the IOSB field.

Before proceeding further, the process should now examine the condition value returned in R0. If the condition value indicates that an error occurred, then the request never made it to the driver. The process should invoke an error handling routine which takes appropriate corrective action based on this condition value.

If no error is indicated, then the process may proceed to do other work. The process should not presume anything about the success or failure of its request until it has explicitly been notified by VMS that its request is complete. This notification occurs at the end of the second phase and will be in the form of the Event Flag being posted or the delivery of the requested AST.

3.5.1.2 Driver Handles $QIO Request

During the second phase, the driver handles the I/O request. The process remains totally oblivious as to how this is done. There are two preferred mechanisms for notifying the process about the completion of its request (i.e. once the driver has done as much with the request as it can).

3.5.1.3 AST Notification

One of the two preferred mechanisms is an AST which the process may have optionally specified when invoking the $QIO. Upon completion of the request (successful or otherwise), VMS delivers the AST to the process. "Somewhat like" a device interrupting the CPU, the AST is invoked asynchronously relative to the normal flow of instructions within the process. VMS builds a call frame on the stack in much the same way as the process would if it had used a CALLS to invoke the AST. The current PC of the process is preserved on the stack and the process finds itself in the AST service routine. Because of how ASTs are delivered, they should generally end with a RET instruction.

3.5.1.4 Event Flag Notification

The second preferred mechanism is the event flag which the process may have optionally specified. This flag is cleared upon entry to the $QIO system service code, and it is set upon completion of the request. The process could periodically poll this flag to see if it is set; but this is wasteful of CPU cycles. If the process can do no further work until the $QIO completes, then it is generally considered preferable that the process request VMS to place it in a wait state until the event flag is set.

The wait can be performed by means of system services such as *$SYNCH* and *$WAITFR*. In selecting an event flag for this purpose, the process should take care that no other events are also using this same event flag.

Once the AST is delivered or the event flag is set, the process should examine the I/O status block whose address was specified by the IOSB parameter. This quadword is cleared prior to the request being handed to the driver. The second condition value returned to the process is stored in the low order 16 bits of this quadword upon completion of the request, and just prior to the delivery of the AST or setting of the event flag.

This second condition value indicates how successful the request was handled after it was passed to the driver.

NOTE

Condition values are 32-bit longwords. However, all condition values returned in an I/O status block have zeros for their high order 16 bits. Only the low order 16 bits of these quantities are actually returned in an I/O status block. It is therefore very common to extract the low order word from the status block and zero extend it into a 32-bit longword before using it.

In the case of a disk read or write operation, the actual number of bytes transferred is returned in bits <47:16> of the IOSB, and bits <63:48> are currently cleared by routine *EXE$FINISHIOC* as of this writing.

3.5.2 What VMS Sees

For read and write requests directed to disks on "MSCP speaking" controllers, the steps taken by the operating system fall into six major phases:

- I/O pre-processing prior to passing the request to DUDRIVER.
- DUDRIVER building an MSCP command describing the request.
- Transmission of the MSCP command to the controller by the SCS and PPD layers.
- Receipt from the controller of an MSCP end message corresponding to the MSCP command.
- DUDRIVER processing the end message.
- I/O postprocessing and AST delivery.

Except for exchanges across the NI, VMS never actually sees the data transfer. Briefly, here is an explanation of why for each type of port:

3.5.2.1 CI and DSSI Ports

The different implementations of the CI port hardware (e.g. CI780, CIBCA, CIXCD etc.) and some DSSI port implementations are *Direct Memory Access* (DMA) devices. They can read and write VAX memory without the direct involvement of VMS.

For a write operation, the MSCP command is sent to the remote controller, which is either an HSC, ISE, or another VAX running the VMS based MSCP server. When the remote controller is ready to accept the data, it sends a message to the local CI/DSSI port hardware requesting the data. The local CI/DSSI extracts the data directly from local VAX memory and transmits it. For a read operation, the local CI/DSSI port writes the data directly into local VAX memory when it receives it from the remote controller.

3.5.2.2 Local Ports

Local DSA controllers (e.g. UDA50s, KDB50s, KDM70s, etc.) are also DMA devices. Once given an MSCP command, they too can extract data directly from or write data directly into local VAX memory.

3.5.2.3 NI Ports

With NI ports, remember that PEDRIVER has a CI *Port Emulator* (PEM) component. Part of the emulation performed by this component is the transfer of data to and from local VAX memory that would otherwise be done by real CI port hardware. Thus, VMS directly sees these transfers, but only to the extent that they pass through the NIDRIVER and the PEM component of PEDRIVER.

Here, then, is a summary of the tasks performed by each of the six major steps.

3.5.2.4 I/O Pre-processing

The I/O pre-processing step is responsible for allocating and initializing the IRP to describe the request. In part, this is consistent with what the process perceives as the first phase of handling a $QIO. The event flag is cleared, the CHAN and FUNC parameters are validated, and the IOSB is probed to see that it is writeable. Then the IRP is allocated and filled in with various quantities such as the ASTADR, ASTPRM, and EFN parameters, the process's PID, and function code. These tasks, however, are all device and function independent.

Next, device and function dependent pre-processing tasks are performed. The data transfer buffer specified by the P1 and P2 parameters is probed for proper read/write access, locked in physical memory, and also mapped to system space. The total requested transfer size (P2 parameter) and starting VBN (P3 parameter) are stored in the IRP. Then the first segment of the transfer is mapped, and the starting LBN and segment size is stored in the IRP.

Now the IRP is queued to DUDRIVER for MSCP specific processing.

3.5.2.5 DUDRIVER Builds MSCP Command

An SCS message buffer is allocated, and in this buffer is stored SCS routing information necessary to send its contents to the controller. MSCP protocol information describing the segment represented by the IRP is constructed and stored in the message buffer. This includes the MSCP unit number, the MSCP op code, and a "buffer handle" by which the controller can access the data transfer buffer in VAX host memory. Then the message buffer is passed to the SCS and PPD routines in PADRIVER (CI), PIDRIVER (DSSI), PEDRIVER (NI), or PUDRIVER (Local) for transmission.

3.5.2.6 Transmission of the Command to the Controller

What happens here is dependent upon the type of port used. In general, SCS code verifies that there is an open connection with the controller's disk server. Next, SCS/PPD header information is inserted into the message buffer. This would include such items as the SCS message length, the fact that this is an application message bound for a remote SYSAP as opposed to a control message to be handled by the controller's SCS layer, and an op code of ("send message" as opposed to "send datagram") for the transmitting port.

The message buffer is handed to the port for transmission, and the request is suspended. The context of the request is saved within the CDRP portion of the associated IRP/CDRP pair.

The contents of the message buffer is transmitted by the port to the controller. When the controller is "ready", the data to be transferred for this segment is exchanged between the controller and local VAX memory. As explained above, this exchange is effectively transparent to the VMS operating system. However, once all the data for this segment has been transferred, the controller releases the MSCP end message corresponding to the data transfer segment.

3.5.2.7 End Message Received from Controller

The port level software verifies that there were no local port hardware errors associated with the reception of the message from the CI, DSSI or NI. Then the end message is passed to SCS code for routing to the disk class driver.

3.5.2.8 Class Driver Processes End Message

Using the RSPID mechanism discussed in the first chapter, DUDRIVER associates the end message with the CDRP containing the suspended context of the request, and then resumes the request. The MSCP status code is checked to see that no errors were reported by the controller. IOSB information is constructed based on the last segment transferred. Various SCS resources are released. Finally, the IRP is passed to I/O postprocessing.

3.5.2.9 I/O Postprocessing and AST Delivery

If more data remains to be transferred for this request, then the IRP is updated to reflect the next segment. This involves setting up a new starting VBN and a new starting LBN, mapping as much of the remaining request as possible into another segment, and specifying a new segment transfer size. Then the IRP is passed back to DUDRIVER.

If the entire request is complete, buffer pages are unlocked, the event flag is set, and AST delivery occurs.

3.6 Details of the Flow of a $QIO

The remainder of this chapter presents in detail the steps involved in VMS handling a $QIO request to read or write a disk on an "MSCP speaking" controller.

First the presentation presumes that the "MSCP speaking" controller is either an HSC or a remote VAX running the VMS based MSCP server, and that the local host is using a CI780 or CI750 port for SCS communication.

Had the local host been using one of the other CI ports, then the name of the interrupt service routine would be different. For example, in the case of the CIXCD, the interrupt service routine would be INTERRUPT_CIXCD. The CI port interrupt service routines all perform functionally the same tasks. They check port specific registers for local port hardware errors associated with the receipt of a packet, and then converge to common code to pass the received packet to its destination within the local host.

For an NI port, it should be remembered that the PEM component of PEDRIVER is emulating the functions of a CI port. It is also interfacing with the NIDRIVER to accomplish the actual transmission and reception on the NI. These additional layers of software exist "beneath" the SCS layer of PEDRIVER and are transparent to the class driver.

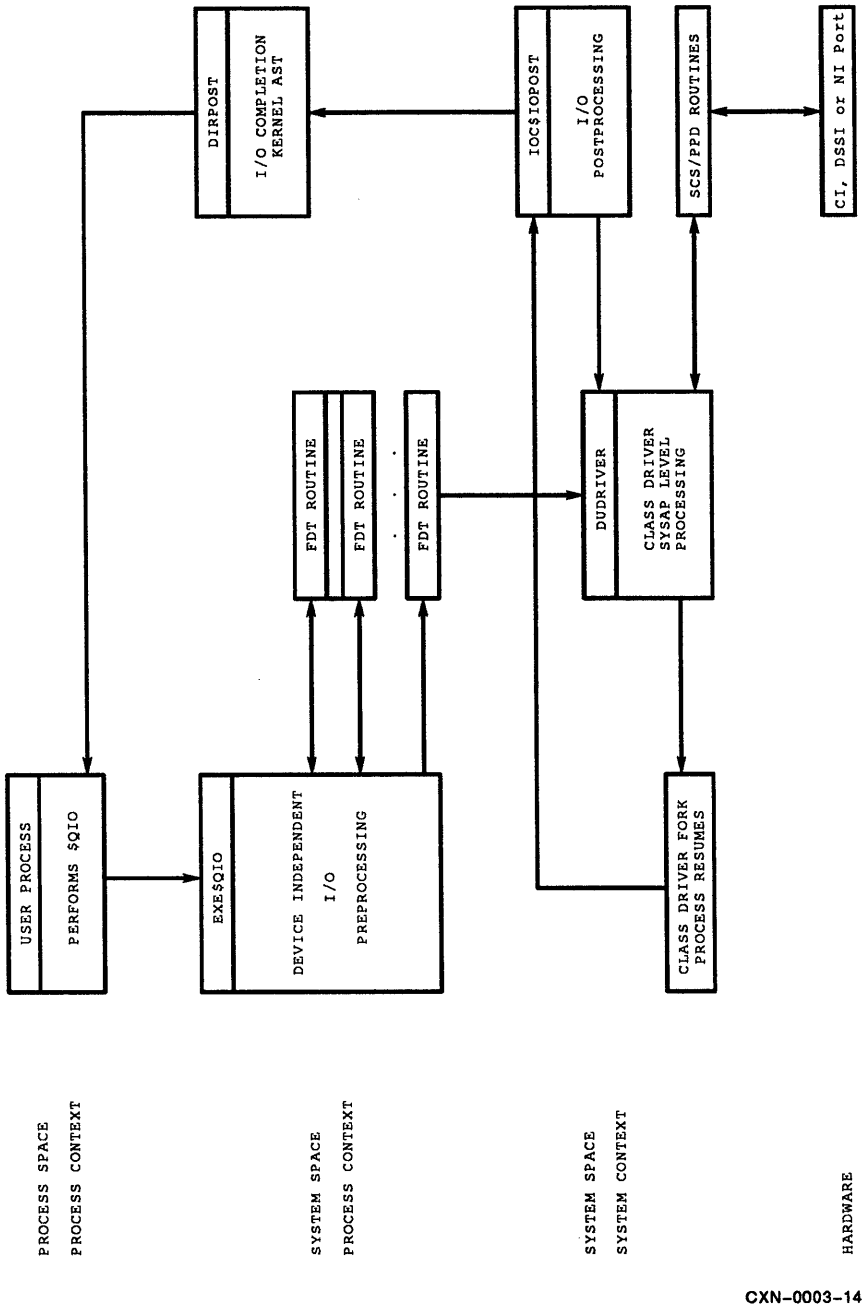
Because of this, PEDRIVER's "pseudo interrupt service" routine, *PE$INT*, has no port registers to check for possible hardware errors. Thus, it promptly branches to its copy of the common code to which CI port drivers would; and from that point on, there is no difference in the flow of what is presented here.

There are some noticeable differences if the $QIO request involves a local DSA controller. These differences are confined strictly to SCS and port driver functionality. They are completely transparent to DUDRIVER and do not affect the overall flow of the $QIO.

For exchanging MSCP commands and end messages with local DSA controllers (UDA50, KDB50, KDM70 etc.), PUDRIVER uses a *Command Ring* in place of the CI port command queues. It also uses a *Response Ring* in place of the CI port response queue. These concepts and their impact are presented after the sections detailing $QIO flow involving CI ports.

It should be remembered that what happens in the port drivers is transparent to DUDRIVER, and beyond the scope of this book. Port driver details presented here are provided merely for the sake of completeness, and for those who may wish to have some understanding of the interactions of a class driver with a port driver. Figure 3–6 displays the general flow of the Qio through the disk class driver.

Figure 3–6: QIO Flow Through the Class Driver



3.6.1 Device Independent I/O Pre-processing

Execution of a $QIO begins with routine EXE$QIO. This routine is primarily responsible for validating the function code and device *Independent* parameters (those other than P1, P2, etc.). EXE$QIO also allocates an IRP/CDRP pair, initializes the device independent fields in the IRP, and then enters the loop for Function Decision Table processing. The following list details the steps taken by EXE$QIO:

- Clears the event flag specified by the EFN parameter.
- Validates the I/O channel number specified by the CHAN parameter.
  - — First, the I/O channel number is range checked by verifying that it is greater than 0 but not greater than the content of *CTL$GW_CHINDX*. (Remember that CTL$GW_CHINDX contains the highest I/O channel number assigned thusfar during the life of the process.)
  - — Then it simultaneously checks that the CCB associated with the I/O channel number is "in use" and that the process is allowed to access the channel.

    This is done by verifying that the access mode of the process at the time it requested the $QIO is numerically less than the content of the *CCB$B_AMOD* field.
- Fetches the address of the UCB from the CCB, the address of the DDT from the UCB, and then the address of the FDT from the DDT.
- Validates the function code supplied as the FUNC parameter against the FDT's "valid I/O function mask". (Since this is not a buffered I/O operation, the second quadword function mask is not used.)
- Examines the status field, UCB$W_STS, to verify that the device is online.
- If an I/O status block was specified using the IOSB parameter, then EXE$QIO makes sure that the status block is writeable and clears it.
- IPL is now set to IPL$_ASTDEL.

  System space data structures are about to be allocated and/or modified based on a request from some process. Therefore, IPL needs to be high enough to block deletion of this process. Process deletion is accomplished by a special kernel mode AST. Raising IPL to IPL$_ASTDEL (or higher) prevents deletion of the process which issued the $QIO request.
- Charges appropriate process quota for transfer
- Calls *EXE$ALLOCIRP* to allocate an IRP from nonpaged pool if IRP lookaside list is empty.

  NOTE

  The process is placed in RSN$_NPDYNMEM resource wait if insufficient non-paged pool is available.

  IPL is temporarily raised to IPL$_SYNCH during this allocation.

- Initializes device and function independent fields in the IRP.
  - — PID (from PCB of process).
  - — AST address and parameter ($QIO parameters ASTADR and ASTPRM).

- — WCB address (from CCB).
- — UCB address (from CCB).
- — Function code ($QIO FUNC parameter).
- — Event flag ($QIO EFN parameter).
- — Process base priority (from PCB).
- — Address of process's I/O status block ($QIO IOSB parameter).
- — I/O channel number ($QIO CHAN parameter).

- At this point, EXE$QIO falls into the loop which calls FDT routines. For standard read requests, there is only one FDT routine: *ACP$READBLK*. For standard write requests, again there is only one FDT routine: *ACP$WRITEBLK*.

  Since there is only a single FDT routine invoked for either read (ACP$READBLK) or write (ACP$WRITEBLK) operations, the FDT routine will pass the IRP to the driver rather than returning to its caller. Once the IRP has been queued to the driver, a branch is taken to return to the process which issued the request.

3.6.2 Device and Function Dependent I/O Pre-processing

Device and function dependent I/O pre-processing involves handling the device and function dependent parameters (P1, P2, etc.), initializing function and device dependent fields in the IRP, and then passing the IRP to the driver.

The FDT routines which perform these tasks for standard read and write requests, ACP$READBLK and ACP$WRITEBLK, differ only in their initial step. They then converge to common code.

- Both routines check the accessibility of the I/O buffer whose address is passed as the value of the P1 parameter. In so doing, they also lock in physical memory the pages containing the buffer. For a request to read from a disk to the buffer, this is done by calling *EXE$READLOCK*. For a write to disk from the buffer, this is done by calling *EXE$WRITELOCK*.
  - — Verifies that the buffer is write accessible if this is a "read from disk" request, or that the buffer is read accessible if this is "write to disk" request.
  - — The buffer is not required to start on a page boundary. Therefore the byte offset into the first buffer page for the actual start of the buffer is stored in the *BOFF* field of the IRP. (This byte offset is merely the low order 9 bits of the P1 parameter.)
  - — Locks buffer pages in physical memory.

    In the process of doing so, consecutive system virtual pages are mapped to this buffer. The system virtual address of the first of the associated consecutive system PTEs (Page Table Entries) is stored in the SVAPTE field of the IRP.

    If necessary, this operation will fault the buffer pages into physical memory.
- Initializes the *OBCNT* field of the IRP to contain the total number of bytes to transfer.

  This is the value of the P2 parameter. However, it is actually copied from the *BCNT* field of the IRP where it was left by routine EXE$READLOCK or EXE$WRITELOCK.

  This is known as the *original byte count* of the transfer.

- Clears the *accumulated byte count* (ABCNT) field, of the IRP, indicating that no bytes have as yet been transferred for this $QIO request.

  If the entire request does not map to a single set of consecutive logical blocks, then it will be broken down into transfer segments. Each of these segments will consist of a set of consecutive logical blocks. As a transfer segment completes, the sum of the size of that segment and the content of the ABCNT field is computed. If this sum is less than the quantity stored in the OBCNT field, then the request is not yet complete.

  The ABCNT field is updated by storing this sum there, and the IRP is recycled through the driver again to transfer the next segment. The segments are processed in ascending order according to the starting VBN of each segment. (This segmentation is often referred to as *split I/O*).
- For virtual I/O functions, checks the IRP to verify that a WCB exists, indicating that the process has "accessed" (i.e. opened) the file. Flags in the WCB are also checked to verify that the process has proper read/write access to the pages mapped by the WCB.
- Sets the VIRTUAL function flag in the IRP's I/O request status field, *IRP$W_STS*.
- Saves the starting VBN of the transfer (i.e. the value of the P3 parameter) in the *IRP$L_SEGVBN* field of the IRP.

  If the request is broken down into segments due to fragmentation of the file, then this field reflects the starting VBN of the segment currently being transferred. Thus, upon completion of each segment, the SEGVBN must be updated to contain the starting VBN of the next segment.
- Calls *IOC$MAPVBLK* to map the starting VBN to an LBN, and also as much of the request as possible.
  - — Searches through the WCB for an entry mapping the starting VBN and determines how much of the transfer maps to consecutive logical blocks. (This depends on the extent of disk file fragmentation.)
  - — Returns starting LBN, number of unmapped bytes, and the address of the "proper" UCB. (If there is a volume set involved, then the UCB whose address is in the CCB is for the root volume. However, the starting VBN may map to some other unit in the set.)
- Address of the "proper" UCB is stored in the IRP at offset IRP$L_UCB.
- Computes the actual number of bytes to transfer (i.e. the length of this segment) as the original byte count less the number of unmapped bytes. The result of this computation is stored in the BCNT field of the IRP.

  Observe at this point that the BCNT field contains only the size of the first segment of the request, whereas the OBCNT contains the total size of the request. These are equal if and only if the entire request mapped to a single set of consecutive logical blocks; thus, there would be only one segment.

  If, however, the request had to be broken into two or more segments due to file fragmentation, then the content of the BCNT field will be less than that of the OBCNT field.
- The starting LBN is stored in the *IRP$L_MEDIA* field of the IRP. (This is done by a call to routine *IOC$CVTLOGPHY* which would convert a logical block number into a physical block number if the disk were not DSA.)

- Branches to routine *EXE$QIODRVPKT* to "queue" the IRP to the driver.
  - — First, EXE$QIODRVPKT calls routine *EXE$INSIOQ* to actually pass the IRP to the driver. Routine EXE$INSIOQ takes out the *FORK spinlock* while it manipulates the I/O queue. Upon return from EXE$INSIOQ, it then branches to *EXE$QIORETURN*, which sets its IPL to 0 and effects a return to the process which issued the $QIO. EXE$QIORETURN also supplies the SS$_NORMAL status returned to the process in R0 (as opposed to the status returned in the IOSB).

NOTE

It is key to observe that all subroutine calls and returns have been done by instructions which merely save and restore the PC, such as JSB/RSB combinations, but do not alter the FP. FDT processing routines, for example, are invoked by a JSB. The return done by EXE$QIORETURN is done by a RET, which makes use of the FP to return to the system service dispatching mechanism. From there, a return is made to the process.

  - — Routine EXE$INSIOQ calls routine *IOC$INITIATE*, which verifies that the operation is allowed on this CPU (check for *affinity*) and then branches to the driver specific start I/O routine whose address is at offset *DDT$L_START* in the Driver Dispatch Table for the driver. The start I/O routine in DUDRIVER is *DU_STARTIO*.

NOTE

EXE$INSIOQ uses the *FORKLOCK* macro to take out the FORK spinlock.

The BSY flag in the STS field of the UCB has no effect with DSA disks since the start I/O routine immediately clears it. This flag pertains only to older disks, such as MASSBUS disks, whose controllers can deal with only one operation per disk at a time.

General Note -

If IOC$MAPVBLK fails to find the necessary mapping information in the WCB as described above, a branch is taken to *EXE$QIOACPPKT* which hands off the IRP to the XQP. (It would pass it to the ACP if the disk had been ODS-1 format.) The XQP performs a *window turn*. Then the XQP proceeds in the same manner as FDT processing would have, had IOC$MAPVBLK been able to map the starting VBN and at least part of the request. The XQP initializes the UCB, BCNT, and MEDIA fields of the IRP and then "queues" the IRP to the driver.

3.6.3 Class Driver SCS Resource Allocation

This is where the $QIO request enters the disk class driver. Here, routine *DU_STARTIO* allocates SCS resources necessary to support the request. These resources are a RSPID and a message buffer in which to build an MSCP command to be sent to the "MSCP speaking" controller. (Remember that the CDRP was allocated as an extension of the IRP.)

- The fork IPL field of the CDRP, CDRP$B_FIPL, is set to contain SPL$C_SCS.

- If the *RWAITCNT* field in the UCB is nonzero, normal I/O requests for this unit are being stalled. Under such conditions, the IRP is queued to the UCB and DUDRIVER will take no further action for this request at this time. The request remains suspended at this point until the RWAITCNT field is reset to 0, indicating that normal I/O on this unit has been resumed.

  One example of this situation would be when the unit is undergoing mount verification.
- The address of the CDT is copied from the UCB into the CDRP.
- A RSPID and associated RDT entry are allocated, and the RSPID is placed in the CDRP.

NOTE

If no RSPIDs are available, the current context is saved in the CDRP, the CDRP is queued to the RDT, and a return is made to the "caller's caller". This facilitates a return being made to the process while leaving the fork thread representing the request suspended. The fork thread is resumed at this point when some other fork thread releases a RSPID and RDT entry, making them available to this thread.

- Allocates from nonpaged pool an SCS message buffer in which to build the MSCP command to describe this request to the "MSCP speaking" controller. This is done by calling routine *FPC$ALLOCMSG* in PADRIVER.
  - — Verifies that there is an open connection with the MSCP server in the controller.
  - — Verifies that there is at least one send credit.
  - — Allocates a buffer from nonpaged pool. The PPD$B_TYPE field of this buffer is set to DYN$C_CIMSG (as opposed to DYN$C_CIDG).
  - — Copies destination CONID from RCONID field of the CDT into the message buffer.
  - — Stores the address of the message buffer in the CDRP at offset CDRP$L_MSG_BUF.
  - — Decrements the send credit field in the CDT.

NOTE

If no send credits are available, the CDRP is inserted into a credit wait queue on the CDT, and the fork thread for this request is suspended at this point until a send credit is available.

If nonpaged pool is unavailable, the CDRP is queued to the PDT's wait queue, and the fork thread is suspended at this point until pool is available.

If the connection with the MSCP server in the controller is not open, the fork thread is effectively "terminated" here.

- Stores RSPID in the message buffer. (The RSPID will serve as an MSCP command reference number in situations such as where the local DUDRIVER must inquire with the controller's server as to the status of the command.)
- MSCP unit number is copied from the UCB to the message buffer.
- Dispatches on the basis of the I/O function code in the CDRP:

  - — *START_WRITEPBLK* - if write operation

— *START_READPBLK* - if read operation

3.6.4 DUDRIVER Builds MSCP Command

Routines START_WRITEPBLK and START_READPBLK differ only in their initial step, and then converge into common code. They have the responsibility for constructing and storing the MSCP protocol information in the message buffer. Then they pass the CDRP containing the address of the MSCP message buffer to the SCS layer, and from there the MSCP command will be transmitted to the server.

- Sets the MSCP op code field in the message buffer to *MSCP$K_OP_WRITE* or *MSCP$K_OP_READ.*
- Maps the IRP by invoking macro *MAP_IRP.*
  - — Removes a buffer descriptor from the linked list of free buffer descriptors in the BDT and initializes the descriptor based on SVAPTE, BCNT, and BOFF fields in the IRP.
  - — Builds the buffer handle in the CDRP.
    - o Transfer offset set to 0.
    - o Buffer name based on sequence number and index of the BDT entry used for the buffer descriptor.
    - o The RCONID is copied from the CDT.

NOTE

If no free buffer descriptor is available, the CDRP is queued to the BDT wait queue, and this driver fork thread is suspended at this point until a free buffer descriptor is available.

- Copies the buffer handle from the CDRP into the SCS message buffer.
- Copies into the SCS message buffer the "byte count to transfer" and starting LBN from the BCNT and MEDIA fields of the CDRP.
- Passes the message buffer to the SCS layer for transmission.
  - — Inserts the CDRP into the queue of active CDRPs on the CDDB associated with the controller.
  - — Executes a JMP *@PDT$L_SNDCNTMSG*({pdt address}) to actually send the buffer.

NOTE

The class driver thread is folded into the CDRP fork block and suspended until the MSCP end message corresponding to this request arrives from the controller. The end message will contain a copy of the RSPID passed to the controller in the MSCP command. The RSPID will be used by the class driver input dispatcher routine to identify and resume this particular thread.

3.6.5 Transmission of Message by SCS and PPD Layers

Routine *FPC$SNDCNTMSG* in PADRIVER inserts SCS header information into the message buffer containing the MSCP command. Next it calls the routine in the PPD layer to insert PPD header information into the message buffer and queue the buffer to the port for transmission. Finally, it suspends the driver fork thread representing this $QIO request.

- Verifies that the SCS connection with the disk server to which the message is about to be sent is still open.
- Clears the register containing the value to be used for the RETFLAG since a RSPID is associated with this message. This will indicate to the port that the message buffer should be returned to the MFREEQ, and not the RSPQ, if no errors occur during transmission.

  (The presence of a RSPID tells the port driver that a response is expected to this message. The port should insert an extra buffer into the MFREEQ in anticipation of receiving the response; and the buffer containing this message is just as good as any other.)
- Increments the PENDREC field in the CDT.
- Establishes the SCS message length in the SCS$W_LENGTH field of the message buffer.
- Sets the SCS$W_MTYPE field in the buffer to SCS$C_APPL_MSG. (This is an application message intended for a SYSAP on the destination node/controller, and not an SCS control message intended for the destination's SCS layer.)
- Copies the content of the PENDREC field in the CDT to the CREDIT field in the SCS header portion of the message buffer to extend pending receive credits to the remote SYSAP. Adds the PENDREC field into the REC field of CDT, and then clears the PENDREC field.
- Copies the local CONID from the CDT into the SCS$L_SRC_CONID field in the message buffer.
- Calls *SCSCI$SNDMSG* to fill in the PPD header and transmit the message.
  - — Sets the PPD$W_MTYPE field to PPD$C_SCS_MSG (as opposed to SCS_DG, START, etc.)
  - — Sets the PPD$B_OPC field to PPD$C_SNDMSG (as opposed to SNDDG, SNDDAT, etc.)
  - — Copies the destination port number from the RSTATION field in the PB to the PPD$B_PORT field in the buffer.
  - — Inserts the setting (0 in this case) of the RETFLAG into the PPD$B_FLAGS field.
  - — Queues the buffer to COMQHIGH (Command Queue 1).
- Clears the MSG_BUF field in the CDRP. (The SCS message buffer has been given to the port; the CDRP no longer "owns" it.)
- Suspends this driver fork thread.
  - — Stores the current contents of R3 and R4, as well as the PC at which to resume this fork thread, in the CDRP.
  - — Inserts the CDRP into the CDRP wait queue (CDDB$L_CDRPQFL) for the controller to which the MSCP command is being sent. The CDRP will remain in this queue until the corresponding MSCP end message is received from the controller's disk server.

3.6.6 End Message Received by PPD and SCS Layers

INTERRUPT_CI780 is the routine in PADRIVER which fields interrupts from a CI780 computer interconnect. It removes the packet containing the MSCP end message from the CI's response queue, and then passes it to the class driver.

- INTERRUPT_CI780 verifies that there are no local CI port hardware errors associated with interrupt produced when the CI inserted the received packet into the RSPQ. Then it calls *SCSCI$FORK* (an alternate name for routine HANDLE_INT).
- HANDLE_INT verifies that no errors are reported in the PPD status field in the received packet and passes it to the SCS layer based on the PPD op code.
  - — Creates a fork process to handle packet(s) in the RSPQ.
  - — Pokes the maintenance timer in the CI.
  - — Removes the entry from the RSPQ and verifies that no errors are indicated in the PPD$B_STATUS field.
  - — Branches to subroutine *SCSCI$PROCESS_RSP_PPD* to process this entry
  - — Routine SCSCI$PROCESS_RSP_PPD branches to REC_MSGREC on the basis of the PPD op code (PPD$C_MSGREC in PPD$B_OPC field).
- REC_MSGREC passes the packet to the SCS layer by branching to SCS$REC_MSGREC.
- Routine SCS$REC_MSGREC does SCS bookkeeping and passes the packet to the disk class driver.
  - — Differentiates this packet from an SCS control message by observing that the SCS$W_MTYPE field contains SCS$C_APPL_MSG.
  - — Verifies that the destination CONID field is valid. First it range checks the 16-bit index portion against the length of the CDL. Then it compares the destination CONID field in the received message with the LCONID field in the CDT pointed to by the index portion of the destination CONID. (If the destination CONID is not valid, the buffer is effectively discarded by being placed in the MFREEQ, and no further processing is done for this packet.)
  - — The CDT$W_REC field (local receive credit, i.e. send credit held by remote server) is decremented.
  - — Credit extended by the remote node (SCS$W_CREDIT in received packet) is added to the local send credit, CDT$W_SEND.
  - — The packet (i.e. buffer containing MSCP end message) is passed to the the disk class driver SYSAP by calling the SYSAP message input routine whose address is in the MSGINPUT field of the CDT.

3.6.7 Disk Class Driver Message Input Dispatching Routine

This routine, *DU$IDR*, is to the disk class driver what an interrupt service routine is to a conventional device driver. The end message is passed here by the port driver (SCS layer).

DU$IDR first verifies that the end message is still "of interest", and then resumes the class driver thread which issued the MSCP command associated with this end message.

- Uses the RSPID to determine if the end message is still "of interest".
  - — Fetches the RSPID from the *MSCP$L_CMD_REF* field in the end message and range checks the index portion of the RSPID. (The maximum value allowed for this index is stored in the RDT$L_MAXRDIDX field of the RDT.)
  - — Using the RSPID, DU$IDR fetches the RDT entry and verifies that the RSPID is still valid. It compares the sequence numbers and checks that the RD$V_BUSY flag is set in the RDT entry.

NOTE

If the end message is no longer "of interest", or if the RSPID is not valid, DU$IDR merely logs an EMB$K_BADRSPID (bad or stale RSPID) error and deallocates the message buffer.

- Fetches the address of the CDRP from the RDT entry, and the address of the CDDB from the AUXSTRUC field of the CDT.
- Compares the CMD_REF field of the end message with the OLDRSPID field of the CDDB. If the end message corresponds to the oldest active command for the CDDB, the OLDRSPID field is cleared.
- The associated class driver thread is resumed by dispatching through the FPC field of the CDRP. (The thread resumes immediately after the point where the MSCP command was passed to the SCS and PPD layers for transmission.)

3.6.8 Class Driver Thread Resumes

Resuming immediately after the SEND_MSCP_MSG macro, the driver thread constructs the information to be returned in the IOSB, releases SCS resources, and branches to the routine to initiate I/O postprocessing.

- Verifies that there was no MSCP error reported in the MSCP end message STATUS field.
- Constructs quadword IOSB information based only on the the segment just completed:
  - — Bits <15:00> are set to contain the status code SS$_NORMAL, indicating success.
  - — Bits <47:16> are set equal to the actual number of bytes transferred by the segment just completed. This quantity is obtained from the BYTE_CNT field of the MSCP end message.

If the the request mapped to a single set of consecutive logical blocks, then the segment just completed represents the entire request. This quantity should be equal to the content of the OBCNT field in the IRP.

If the request involves more than one segment, then this quantity is less than the content of the OBCNT field since the segment just completed represents only part of the request.

— Bits <63:48> are set to 0.

- Calls DUTU$DEALLOC_ALL to release SCS resources held by the CDRP.
  - — UNMAPs the buffer (and resumes any CDRPs waiting for BDT entries). This is done by calling routine *SCS$FPC_UNMAP* to release the buffer descriptor in the BDT.
  - — Deallocates/releases the message buffer containing the end message.
  - — Releases the RSPID (and resumes any CDRP queued to the RDT and waiting for a RSPID).
- Branches to IOC$ALTREQCOM to initiate I/O postprocessing.
  - — Stores IOSB quadword constructed above into the MEDIA field of the IRP.

NOTE

The IRP$L_MEDIA and IRP$L_IOST1 fields are overlays of each other; and the IRP$L_IOST2 field immediately follows the IRP$L_IOST1 field. Thus, if the request was not segmented, the IOST1 and IOST2 longwords have been loaded with the final I/O status information to be later transferred to the process's IOSB.

  - — Inserts the IRP into the I/O postprocessing queue *IOC$GQ_POSTIQ*.
  - — Generates an *IPL$_IOPOST* software interrupt.
  - — Terminates this class driver thread.

3.6.9 I/O Postprocessing and AST Delivery

Invoked by an IPL$_IOPOST software interrupt, routine IOC$IOPOST determines if the entire request is complete. If the request was segmented but is not yet complete, then it adjusts various IRP fields to reflect the next segment and passes the IRP back to the class driver. If the request is complete, IOC$IOPOST performs all appropriate I/O completion activity.

- Removes the IRP from the I/O postprocessing queue.
- Determines if the entire requested I/O transfer is complete.

  The number of bytes transferred by the most recent segment was just stored in the IOST1 and IOST2 fields of the IRP by routine IOC$ALTREQCOM. This quantity is compared with the quantity in the OBCNT field.
  - — If the two quantities are equal, then the request is complete. This could only be true if the request did not have to be broken up due to file fragmentation into multiple segments.

— If the two quantities are not equal, then the number of bytes transferred by the most recent segment represents only part of the entire request. The accumulated number of bytes transferred is updated by adding the number of bytes transferred by the most recent segment into the IRP$L_ABCNT field. Then, the new content of the ABCNT field is copied to bits <47:16> of the quadword made up of the IOST1 and IOST2 longwords.

If the contents of the ABCNT and OBCNT fields are now the same, then the most recent segment was the last and the request is complete. If not, then the request is not complete and there is at least one more segment to transfer.

- If the I/O request is not complete, then IOC$IOPOST does the following:
  — Adjusts the *IRP$L_SEGVBN* to contain the starting VBN of the next segment by adding the number of blocks just transferred into the SEGVBN field.
  — Adjusts the IRP$L_SVAPTE field to contain the system virtual address of the system PTE pointing to buffer page corresponding to beginning of the next segment.
  — Calls IOC$MAPVBLK as before to map as much of the remaining transfer as possible into this next segment. The actual number of bytes to transfer by this next segment and associated starting LBN are stored in the BCNT and MEDIA fields of the IRP.

NOTE

The IRP now represents the next segment in the request.

  — Routine *EXE$INSIOQ* is called as before to pass the IRP to back to the disk class driver (routine DU_STARTIO) again.
- If the I/O request is complete, then IOC$IOPOST takes the following steps:
  — Buffer pages are unlocked and the associated system virtual PTEs released.
  — The event flag specified by the $QIO parameter EFN is now set.
  — Using an IPL$_ASTDEL software interrupt, a kernel AST (address = DIRPOST) is delivered to the process to write status to the IOSB and deliver any user specified AST to the process.

3.7 Impact on $QIO Flow Due to Local DSA Controller

There are some significant differences in the flow of a $QIO when a local DSA controller is involved, instead of a CI, DSSI or NI port. The remainder of this chapter provides an overview of these differences.

For all intents and purposes, these differences are transparent to the disk class driver. They depend upon the type of port used to support SCS communication with the DSA controller, and the internals of that controller. As such, these differences are confined to port driver routines invoked by DUDRIVER. The port driver for local ports is *PUDRIVER*.

3.7.1 Allocating an SCS Message Buffer

The first noticeable difference is when DUDRIVER calls the routine to allocate an SCS message buffer in which to build an MSCP command. These message buffers are allocated from a different pool of buffers than for that of a remote port.

During controller initialization, PUDRIVER sets up a pre-allocated pool of buffers within the Port Descriptor Table (PDT) for each local DSA controller. These buffers are used to exchange MSCP commands and end messages between itself and the controller. The number of these buffers is dependent upon the type of adapter being used with the count being stored at offset *PDT$L_NO_BUFFS* within the PDT. The size of each of these buffers is found at offset *PDT$L_UDAB_LEN*.

3.7.1.1 Ring Buffer Count Calculation

The number of buffers is calculated in routine *BUILD_PDT* based on the size of both the command ring and response ring plus some padding to handle stalled requests (VMS V5.5 specifies 8 additional buffers). The command and response ring sizes are derived from an array of entries contained in (*RINGEXP_ARRAY*) which is indexed by the adapter type. The default number of each buffer type is 2**4 buffers for VMS V5.5. The KDM70 specifies 2**5 buffers when the array is created using the *Create_device_entry* macro.

The size of the buffers is calculated using another array of entries contained in (*MSGLENGTH_ARRAY*). This array is also indexed by the adapter type. The buffer header overhead (20 bytes) is added to the base message size (default 80 bytes) to provide the total size of a *UDAB buffer*. The message text within the UDAB buffer is pointed to by the *UDAB$T_TEXT* offset. The KDM70 specifies a base message size of 108 bytes when the array is created using the Create_device_entry macro.

The address of the start of these buffers is stored at offset *PDT$L_BUFARY* within the PDT. They are indexed by buffer number starting with 0 and going up to PDT$L_NO_BUFFS - 1.

NOTE

While these buffers were originally structured for use with the UDA50, they are actually used with all controllers handled by PUDRIVER.

At offset *PDT$L_CRCONTENT* is an array of longwords, each of which will contain the starting address of one of these buffers in the command ring when required. At controller initialization, these are preset to contain a value of minus one representing an unused entry.

At offset *PDT$L_RRCONTENT* is an array of longwords, each of which will contain the starting address of one of these buffers in the response ring. At controller initialization, these are preset to contain the address of the first ringexp_array (2**4 or 2**5) number of buffers based on the adapter type.

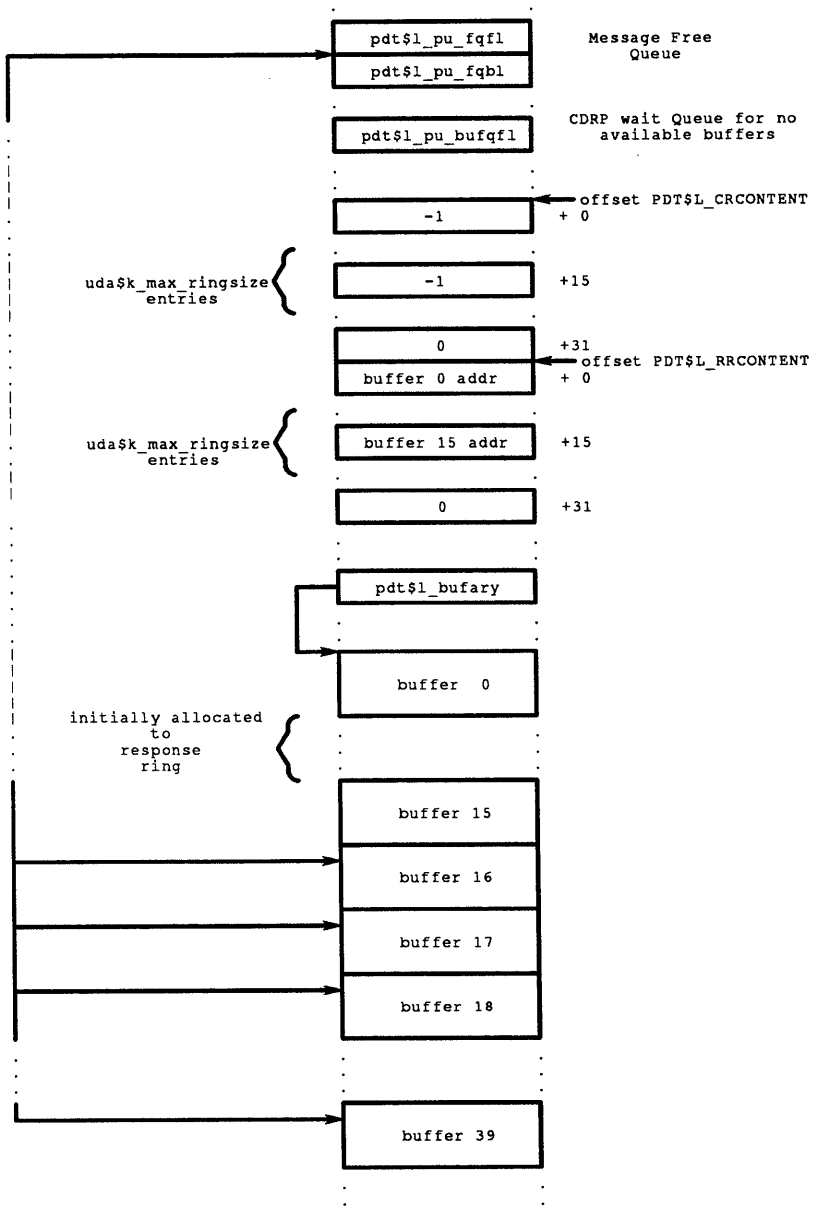
Both of these lists are allocated with *UDA$K_MAX_RINGSIZE* entries regardless of the type of controller being handled. For VMS V5.5, this value equates to 2**5 entries.

When the PDT is initialized, the remaining portion of these buffers are placed in a message buffer free queue whose head is at offset *PDT$L_PU_FQFL*.

For VMS V5.5, the default number of buffers set up in this way is 40 [1], and the number set aside in the response ring is 16. Figure 3–7 is based on these values.

[1] The number of buffers for the KDM70 is 72

Figure 3–7: Local Port Buffer Initial Layout



When DUDRIVER allocates an SCS message buffer to build an MSCP command for a local DSA controller, it does not go to nonpaged pool as it would with a CI. Instead, it calls the routine *FPC$ALLOCMSG* in PUDRIVER to fetch one from the message buffer free queue in the controller's PDT.

If this queue is empty, it will try to reclaim one from what is known as the *command ring*, (another concept to be explained a bit later). If it still can't get one, then the $QIO request is suspended until a message buffer becomes available.

Suspending the request in this case consists of inserting the CDRP representing the request in a PDT message buffer wait queue *(PDT$L_PU_BUFQFL)*. The PC at which to resume the request, namely within this allocation routine, is saved in the CDRP. When some other request relinquishes a buffer to the queue, then the suspended request at the head of the wait queue is resumed.

If the request succeeds in acquiring a buffer, the address of the *text portion* of the buffer is stored in the CDRP at offset *CDRP$L_MSG_BUF*. The text portion of the buffer begins where the first byte of the MSCP command would be stored.

3.7.2 Mapping the IRP

Assume that DUDRIVER was able to obtain the message buffer. The steps used by DUDRIVER to build the MSCP command are the same as in the CI case, except for mapping the IRP. This is done by DUDRIVER calling the appropriate routine in PUDRIVER based on the following table:

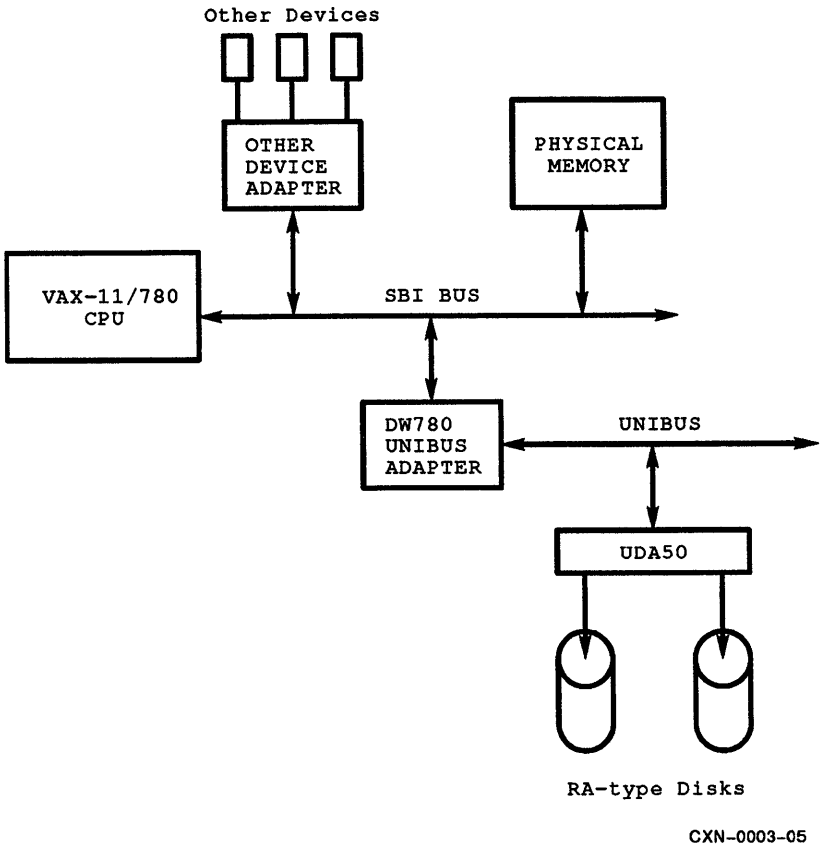
Routine	Application
FPC$MAPIRP	Map a user buffer
FPC$MAPIRP_UV2	Map a buffer for uVax II
FPC$MAPIRP_UV1	Map a buffer for uVax I
FPC$MAPIRP_BDA	Map a buffer for BDA
FPC$MAPIRP_KDM	Map a buffer for KDM

3.7.2.1 The Case of the UDA50

First, consider the UDA50. This is a *UNIBUS* device. Consequently there exist UNIBUS adapters to interface a UNIBUS with the main bus structure of a VAX CPU. For example, the DW780 UNIBUS adapter serves this purpose on VAX-11/780s and VAX-11/785s; and the DW750 serves the same purpose for a VAX-11/750.

Figure 3–8 illustrates the relationship between the UDA50 and the rest of the VAX-11/780. (Of course, there can be many more device adapters and devices than are shown here.)

Figure 3–8: Vax 11/780 Adapter Configuration



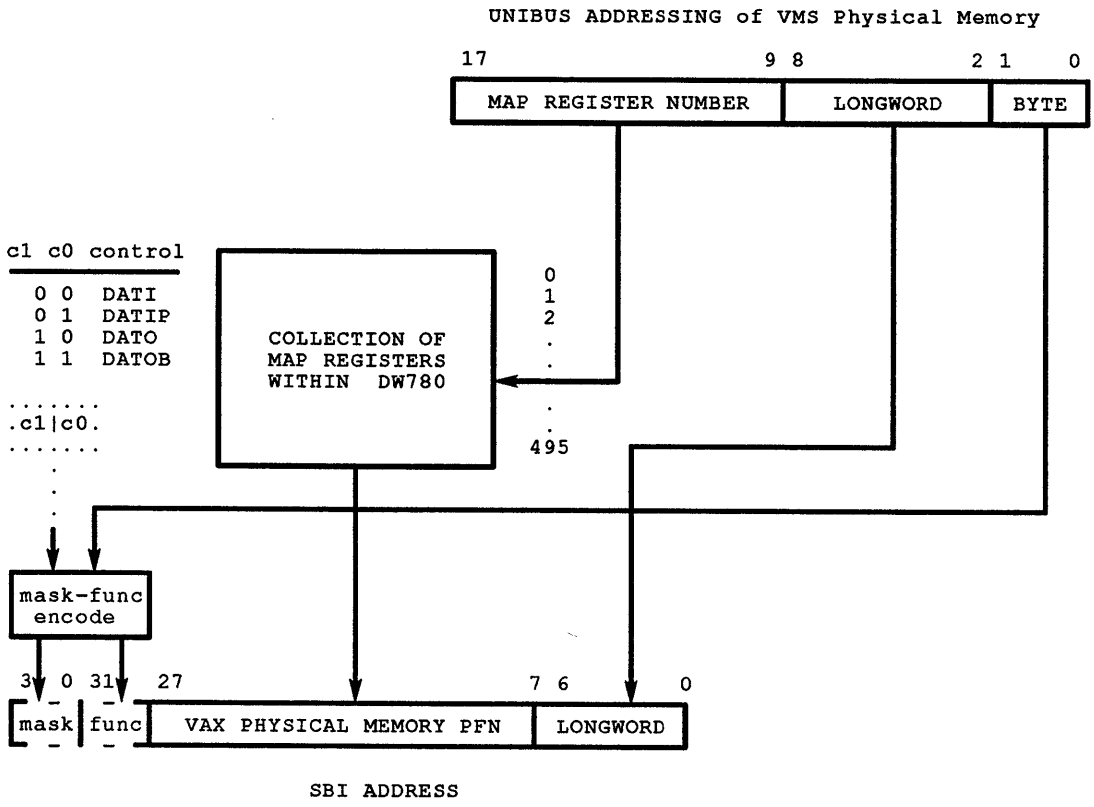
A UNIBUS address is composed of only 18 bits, whereas the 780's *Synchronous Backplane Interconnect* (SBI) supports 28-bit addressing. One of the roles of the DW780 is to translate from an 18-bit UNIBUS address to a 28-bit SBI address whenever one of the devices on the UNIBUS wishes to read or write VAX memory. Here is a brief and very simplified explanation of how this is done.

As Figure 3–9 indicates, within the DW780 is a collection of 496 *mapping registers* which facilitate this translation. The high order 9 bits of a UNIBUS address serve as an index to select one of the mapping registers. From this register comes the SBI page address, which is actually a VAX *Page Frame Number* (PFN) identifying the page of VAX physical memory being referenced. Bits <8:2> identify a longword within that page.

If all transfers between VAX memory and UNIBUS devices were longword aligned, this would be sufficient. However, they aren't. So given the longword in physical memory identified by the 28-bit SBI address, how does the SBI addressing logic determine if the transfer begins with the high order word, or the low order word, within that longword? The answer is UNIBUS address bit 1.

If this bit is set, then the transfer begins with the high order word. If this bit is clear, then the transfer begins with the low order word.

Figure 3–9: Unibus to SBI Mapping



See the VAX maintenance handbook (EK-VAXV2_HB-003) for additonal information.

CXN-0003-06

Getting down to a particular byte provides a special problem. This is because the UNIBUS does not use address bit 0 for addressing, but rather for control. UNIBUS addresses are always on word boundaries. To solve this problem, a special bit, called the *BYTE OFFSET bit* , is provided in each of the 496 map registers. If this bit is set, the transfer begins with an "odd" byte; but if this bit is clear, then the transfer begins with an "even" byte.

In essence, the BYTE OFFSET bit and UNIBUS address bit 1 combine to form an offset relative to byte 0 of the longword specified by the SBI address.

As an example, assume that 32 bits have been assembled in the DW780 for transfer to VAX memory, and that the map register selected by UNIBUS address bits <17:9> contain the PFN 1000. Further assume that UNIBUS address bits <8:2> are all 0.

If the BYTE OFFSET bit is 0 but UNIBUS address bit 1 is a 1, then bytes 0, 1, 2, and 3 of the data will be written to bytes 2, 3, 4, and 5, respectively, of physical memory page 1000. However, if both the BYTE OFFSET bit and UNIBUS address bit 1 are both set, then bytes 0, 1, 2, and 3 are written to bytes 3, 4, 5, and 6, respectively, of physical memory page 1000.

NOTE

As stated earlier, this is a very simplified explanation. Consider that the example just presented would really involve two UNIBUS transfers and two SBI transfers. Thus, for further detail, the reader is referred to the VAX-11/780 *DW780 UNIBUS Adapter Technical Description.*

In this way, the DW780's map registers provide translation between the UNIBUS addressing used by the UDA50 and the SBI physical addressing used by the 11/780 CPU and memory. Now consider how these map registers are used by PUDRIVER's routine FPC$MAPIRP (or FPC$MAPIRP_xxx) to map a segment of a $QIO request represented by an IRP/CDRP pair. This is performed in three major steps:

- First, *IOC$REQMAPUDA* is called to allocate enough map registers to map this segment. One register is allocated per page of data to be transferred. These registers must be consecutive. Also, one additional register is allocated to denote the end of this set of map registers. The byte count (BCNT) and buffer offset (BOFF) fields from the CDRP are used to compute the number of registers allocated.

  NOTE

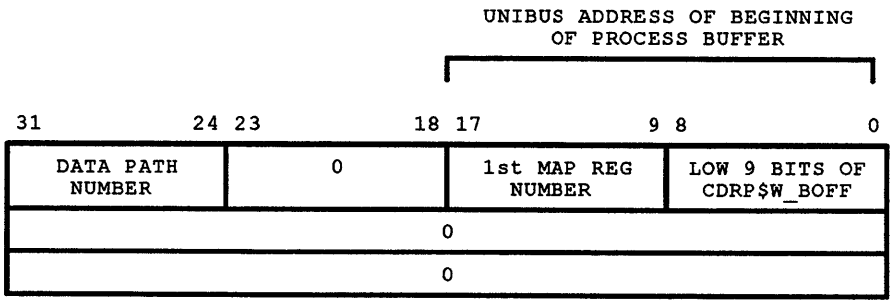
  If enough consecutive registers are not available, the CDRP is placed in a map register wait queue until they are. This queue is in a data structure not covered in this book called an *Adapter Control Block* (ADP).

- Next, *IOC$LUBAUDAMAP* is called to load the map registers for this segment. This step involves two tasks:
  - — A *data path* must be selected. This is basically a 32-bit buffer within the DW780 wherein two 16-bit UNIBUS transfers can be accepted from the UDA50, and then assembled into a single 32-bit longword before being passed to the SBI. This is also where a 32-bit longword can be buffered and broken into two 16-bit UNIBUS transfers destined for the UDA50. The DW780 has 15 such data paths. (There is one more; but this is used for single word exchanges between the UNIBUS and the SBI.)
  - — Then the map registers allocated by routine IOC$REQMAPUDA are loaded. Into each map register is placed the PFN corresponding to one of the process's buffer pages, and also the data path just selected. If the BOFF field of the CDRP indicates that the transfer is going to start with an "odd" byte, then the BYTE OFFSET flag is set in each of these registers.

    The extra map register allocated at the end of the set is cleared to 0.
- Finally, the local buffer handle is constructed and stored in the MSCP message buffer, the address of which is found at offset CDRP$L_LBUFH_AD in the CDRP.

Like the buffer handle that would be used had the port been a CI, this buffer handle also consists of three longwords. But they have a very different format. In fact, the second and third longwords both contain zero. See Figure 3–10 for an illustration of the buffer handle.

Figure 3–10: Buffer Handle for a UDA buffer

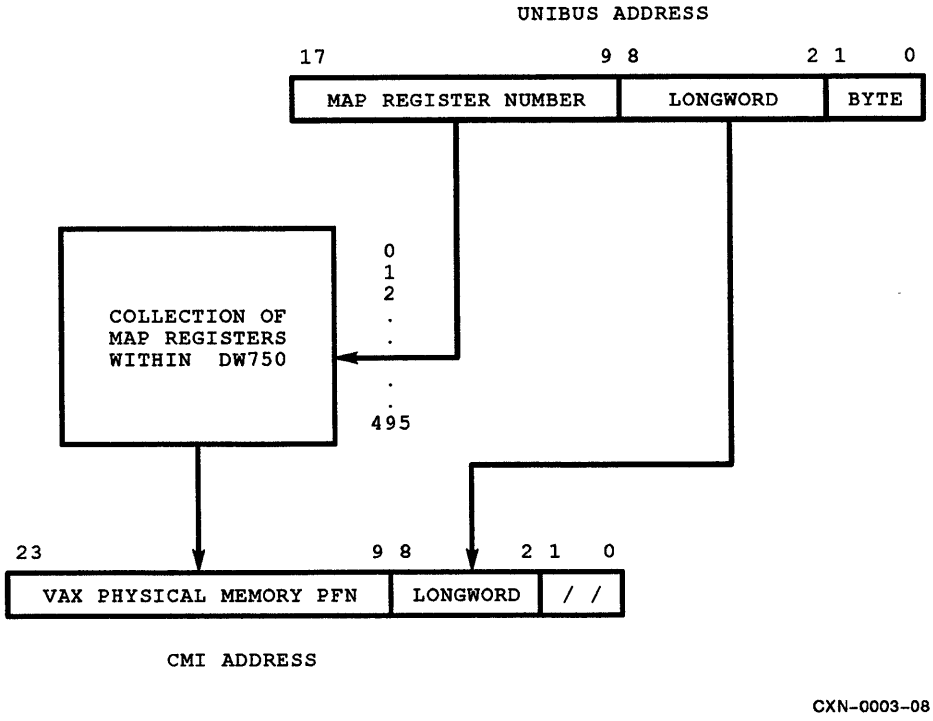


The DW750 is the adapter for interfacing the UNIBUS with the 32-bit *Cpu Memory Interconnect* (CMI) bus on the VAX-11/750. VMS speaks to the UDA50 via the DW750 in the same way as it would via the DW780. However, there are two noticeable differences.

- First, the DW750 provides only three buffered data paths for the conversion between a single 32-bit CMI data transfer and two 16-bit UNIBUS data transfers. (The DW750 does have a fourth data path to facilitate single word exchanges between the CMI and the UNIBUS.)
- The CMI supports only 24-bit physical addresses. So while UNIBUS address bits <8:2> still match up with CMI address bits <8:2>, the map registers supply only 15-bit PFNs to be used as CMI address bits <23:9>.

  Figure 3–11 illustrates UNIBUS to CMI address conversion.

Figure 3–11: Unibus to CMI Mapping



3.7.2.2 Other DSA Controllers

With other DSA controllers, such as the KDB50 on the BI bus or the KDA50 on the microVAX Q-BUS, the steps for mapping the IRP are similar, but with some notable exceptions.

The *Q-BUS* does not allow DMA transfers to or from odd byte aligned buffers. If the buffer is not word aligned, then the data is copied into a page aligned buffer in non-paged pool first. Thus, a direct I/O is essentially turned into a buffered I/O. Also, with the KDA50, Q-bus map registers are used to map transfer buffers.

The map registers used by PUDRIVER for the KDB50 are also quite different. In fact, PUDRIVER refers to them as *pseudo map registers*. While initializing the PDT associated with the port for each of these controllers, PUDRIVER allocates 4 physically contiguous pages of memory. Since each page contains 512 bytes, that's a bit more than enough to hold 496 contiguous longwords of physical memory which will serve as these pseudo map registers.

When mapping the IRP, these pseudo map registers are allocated and loaded by the very same routines that map an IRP in the case of a UDA50. The buffer handle loaded into the SCS message buffer containing the MSCP command has the same format as with the UDA50. It can be concluded that these controllers process what appear to be UNIBUS map registers and

MSCP commands in functionally the same way as a UDA50. However, the concept of buffered data paths do not apply.

3.7.3 Transmission of SCS Message Buffer Containing MSCP Command

In the case of CI port, SCS message buffers ready for transmission are placed in a port command queue within the port's PDT. Given a port for a local DSA controller, the role of a command queue is replaced by that of a command ring.

3.7.3.1 Use of the Command Ring

The command ring consists of a set of consecutive longwords, called descriptors, which are used in a *round robin* fashion. The number of descriptors in the command ring is kept in the PDT offset *PDT$L_RINGSIZE* and is based on the type of controller being configured. (As of VMS V5.5, the value for most controllers is 16 and is 32 for the KDM70.)

These descriptors begin at the address pointed to by offset *PDT$L_CMDRING*. The index of the next available descriptor to use is kept at offset *PDT$B_CRINGINX* and the count of currently used descriptors is kept at offset *PDT$B_CRINGCNT*. Figure 3–12 illustrates the Command Ring.

Figure 3–12: Command Ring Format

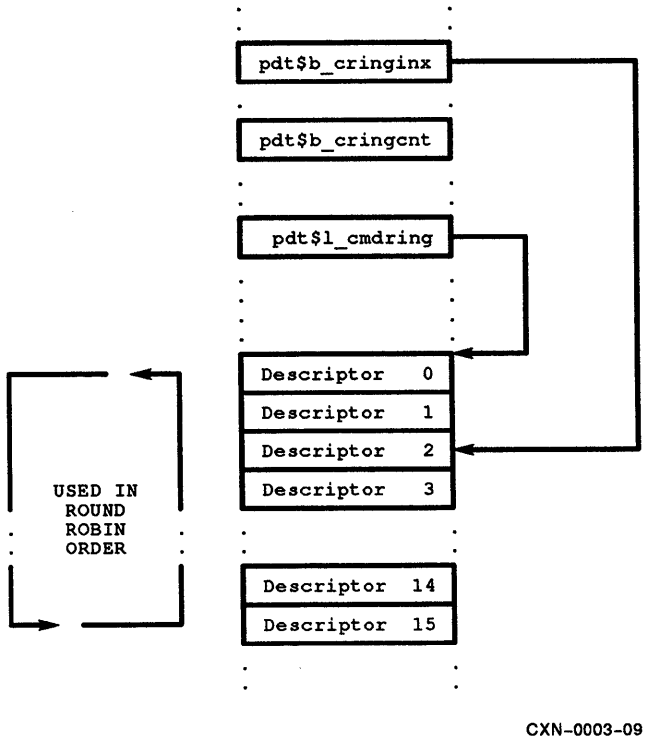
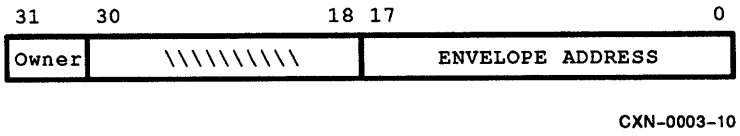


Figure 3–13 illustrates the format of each descriptor:

Figure 3–13: Command Ring Descriptors



The "Owner" bit indicates the ownership of this descriptor. When the host sets up a descriptor for processing by the controller, the host sets the "Owner" bit. When the controller returns ownership of the descriptor back to the host, the controller clears the "Owner" bit.

The *ENVELOPE ADDRESS* field is set by the host to contain the UNIBUS address of the beginning of the text portion of a buffer being passed to the port. Unibus mapping of the buffers is accomplished at port initialization time in routine *INIT_UDA_BUFFERS* and is stored at offset *UDAB$L_DESCRIP* within each buffer.

The format of each SCS message buffer is detailed in Figure 3–14 and described in Table 3–2.

Figure 3–14: Local Port SCS Message Buffer Format

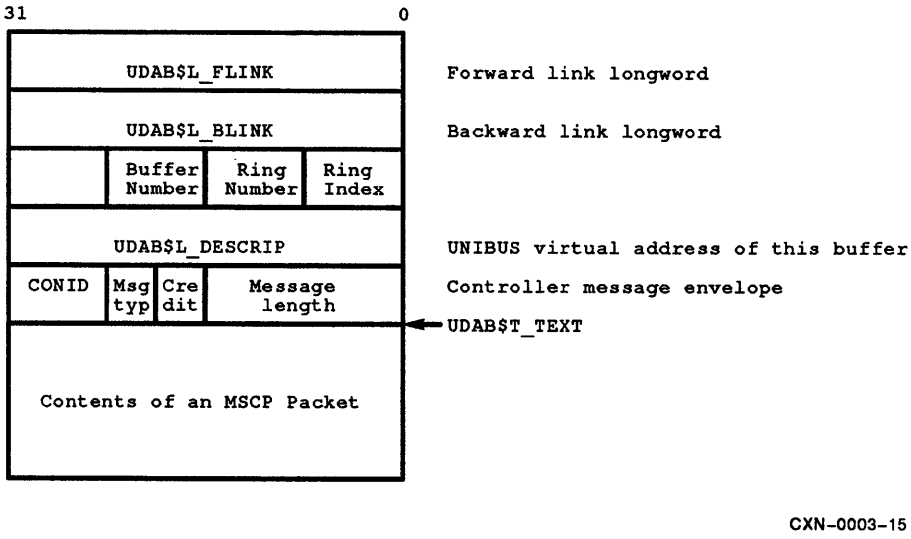


Table 3–2: SCS message buffer fields

Field	Description
1	A FLINK and a BLINK for queuing the buffer on the free queue and also on the SEND Q.
2	UDAB$B_RINGINX which contains the index into a ring on which this buffer has been placed (valid only if the buffer is NOT on the free queue).
3	UDAB$B_RINGNO which contains the number (0 => command ring and 1 => response ring) of the ring on which the buffer is currently residing.
4	UDAB$B_BUFFNO which contains the number of this buffer. There are PDT$L_NO_ BUFFS in total, and they are numbered from zero to PDT$L_NO_BUFFS–1.
5	UDAB$L_DESCRIP which contains the UNIBUS virtual address of the text portion of this buffer in the low order 30 bytes of this longword, and which also has the two high order bits (ownership and full bits) set. This is the precise value that must be placed in a ring longword so as to present the buffer to the controller.
6	The controller header which contains a word of length (of the following text portion only), a byte containing two four bit fields encoding the credit field and the message type field, and a byte of Connection ID.
7	The message text portion.

When DUDRIVER wants to pass the SCS message buffer containing the MSCP command to the local DSA controller, it calls PUDRIVER's routine *FPC$SNDCNTMSG*. This routine does the following:

- First, it verifies that the connection is still open.
- It then verifies that a descriptor is available in the command ring by examining the *PDT$B_CRINGCNT* field of the PDT. This field contains the number of command ring descriptors in use. If this number is less than the value found at offset PDT$L_RINGSIZE, then at least the next descriptor is available. If not, then the SCS message buffer will be left queued to the PDT's queue of "backed up" buffers until one is available. (The head of this queue is offset *PDT$L_PU_SNDQFL*.)
- Next, it places the buffer address into the CRCONTENT array in the PDT at the offset indicated by the PDT$B_CRINGINX field.
- It then copies the UNIBUS address of the "text portion" of the SCS message buffer into the next available command ring descriptor, i.e. the descriptor pointed to by the PDT$B_CRINGINX field.

  The text portion of the SCS message buffer begins with the first byte of the actual MSCP command. The UNIBUS address of the text portion of each such buffer was stored within the buffer at offset UDAB$L_DESCRIP when the PDT for the port was initialized.

  As part of this step, the host also set the "Owner" bit in the descriptor.
- Then, FPC$SNDCNTMSG reads/writes the IP (Initialization and polling) register for the port. This has the effect of "waking up" the controller's microcode to the fact that there is something for it to do in the command ring.

NOTE

Dependent upon the type of controller, either a write to the IP register will force the port to poll the command ring or a read from the IP register will. The low order bit of the *PDT$L_PU_PORTCHAR* field in the PDT will determine whether a read (low bit clear) or write (low bit set) should be performed.

- Finally, it sets the PDT$B_CRINGINX field to point to the next descriptor, and it increments the PDT$B_CRINGCNT field.

3.7.3.2 Reclaiming Descriptors and Buffers from the Command Ring

Once the controller has copied the MSCP command from the message buffer pointed to by the command ring descriptor to its own internal storage, it returns ownership of the descriptor back to the host. In order for the host to reclaim such buffers, it must "poll" the command ring for these descriptors. Since descriptors are released in sequence, the host need only traverse the ring until it finds one still owned by the port.

If a descriptor is found to have been returned to the host, then any buffer queued to the PDT's message buffer send queue, *PDT$L_PU_SNDQFL*, has priority for obtaining it.

If there are no waiting message buffers, then the buffer pointed to by the command ring descriptor is given to the response ring if that ring is not full. The response ring is the mechanism whereby the controller returns MSCP end messages corresponding to commands it received through the command ring.

The response ring is structured the same way as the command ring. However, PUDRIVER endeavors to keep a buffer assigned to every one of its response ring descriptors at all times. This is so that the host is always prepared to receive an incoming message from the controller.

If there are no waiting message buffers and if the response ring is full, then the buffer is placed in the queue of free message buffers, PDT$L_PU_FQBL.

There are three times when this polling is done in attempt to "shake loose" some buffers:

- When FPC$ALLOCMSG is called to allocate a buffer from the queue of free message buffers, but the queue is empty.
- When FPC$SNDCNTMSG is called to send a message, but finds no free descriptors in the command ring.
- When routine *POLL_RSPRING* removes a buffer from the response ring to give it to a class driver. It attempts to replace this buffer with another from the message buffer free queue. If that queue is empty, then this routine is called in an attempt to reclaim one for that purpose.

3.7.4 Receiving MSCP End Message from a Local DSA Controller

The *Response Ring* is used by the port to pass messages from the controller to the VAX host. Starting at offset *PDT$L_RSPRING* in the port's PDT, this ring is structured the same as the command ring. It consists of a set of consecutive descriptors which have the same format as those in the command ring. The number of descriptors in the response ring is kept in the PDT at offset *PDT$L_RINGSIZE*. (As of VMS V5.5, the value for most controllers is 16 and is 32 for the KDM70.)

Unlike command ring descriptors, each response ring descriptor is given one of the pre-allocated buffers during PDT initialization.

The response ring is traversed by the port in a "round robin" fashion. When a local DSA controller wishes to pass an MSCP end message to the host, it copies the data into the buffer whose UNIBUS address is in the next available response ring descriptor (pointed to by *PDT$B_RRINGINX*. It then releases ownership of the descriptor to the host by clearing the "Owner" bit of the descriptor. Finally, it generates a hardware interrupt at IPL 21.

PUDRIVER's interrupt service routine, *PU$INT*, calls routine POLL_RSPRING to examine the response ring looking for descriptors that have been released to the host. The port releases descriptors in sequence; and ownership of a descriptor is returned to the port as soon as the host is finished processing it.

Routine POLL_RSPRING need only traverse the response ring until it encounters the first descriptor not owned by the host. The *PDT$B_RPOLLINX* field contains the index of the next response ring descriptor to be considered by POLL_RSPRING.

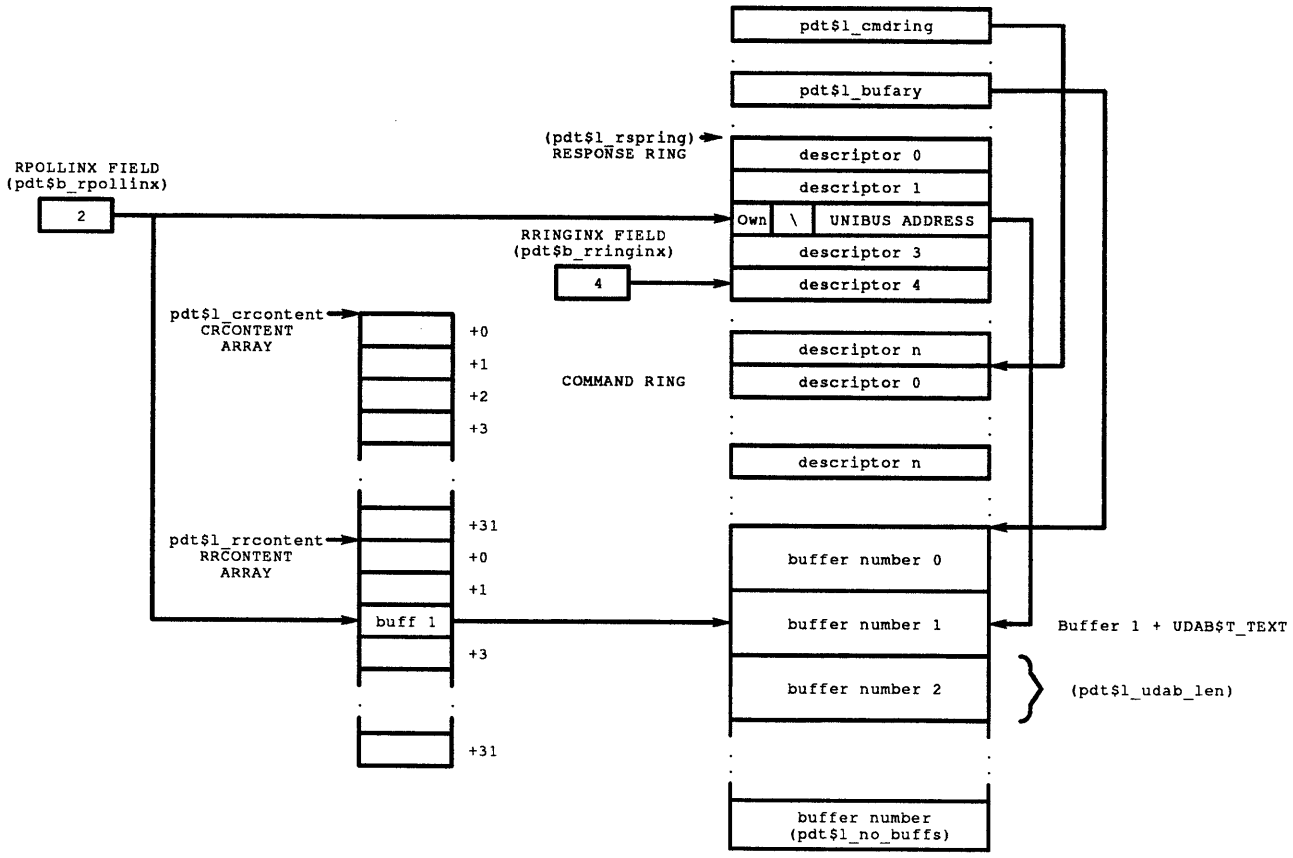
In essence, the host uses routine POLL_RSPRING to logically "chase" the port around the ring. As the port fills buffers and releases their associated descriptors to the host, the port gets further ahead of the host. As the host processes descriptors, the host catches up with the port.

The next illustration shows that routine POLL_RSPRING obtains the index of the first response ring descriptor released in sequence to the host from the PDT's RPOLLINX field. This response ring index is also used as an index into an array of longwords beginning at offset PDT$L_RRCONTENT.

There is a one to one correspondence between the entries in this array and the response ring descriptors. As a descriptor is given a buffer by placing the *UNIBUS address* of that buffer in the descriptor, the buffer address is also stored in the corresponding longword of the RRCONTENT array.

For example, if PDT$B_RPOLLINX contains the number 2, then it is referencing descriptor number 2 in the response ring. It is also referencing the second longword in the PDT$L_ RRCONTENT array. The RRCONTENT array provides the address of the buffer containing the MSCP end message from the controller. Figure 3–15 illustrates the Response Ring configuration:

Figure 3–15: Response Ring Buffer Pointers



The message buffer containing the MSCP end message is then passed to DUDRIVER. However, the mechanism for passing this buffer to DUDRIVER is different from what is employed for CI, DSSI and NI ports. The low order 2 bits of the destination CONID in the buffer are used as an index into a table of longwords beginning at offset *PDT$L_PU_CDTARY* in the Port Descriptor Table for the controller.

This table contains the address of CDTs representing connections between SYSAPS in the local host and SYSAPs in the controller. In particular, the address of the CDT for the connection between DUDRIVER and the controller's disk server is fetched. From the CDT is extracted the address of DUDRIVER's message input routine, and the message buffer is passed to that routine.

It is undesirable that the response ring descriptor containing the UNIBUS address of the message buffer just passed to DUDRIVER be unavailable to the port while DUDRIVER handles the message. During the process of passing the buffer to DUDRIVER, a new buffer is given to the descriptor. A message buffer is removed from the buffer free queue (PDT$L_PU_FQFL) if this queue is not empty; otherwise, one is reclaimed from the command ring

(routine POLL_RSPRING). The UNIBUS address of this buffer is then stored in the response ring descriptor.

3.7.5 Deallocating the SCS Message Buffer

DUDRIVER calls routine *FPC$DEALLOMSG* in PUDRIVER to release an SCS message buffer in which it received a packet from a local DSA controller. This routine decides whether to give the free buffer to the response ring, or to the message buffer free queue (PDT$L_PU_FQFL). The response ring has priority; it is selected over the free queue whenever it is not completely full.

When inserting a buffer on the message buffer free queue, FPC$DEALLOMSG may find that the queue is otherwise empty. If so, there may be requests suspended because this queue was empty when those requests needed buffers in which to build MSCP commands. Thus, in this case, FPC$DEALLOMSG enters code to resume such requests. Their context will be found saved in CDRPs in the message buffer wait queue (PDT$L_PU_BUFQFL).

Chapter 4

Disk Class Driver Error Handling and BUGCHECKs

4.1 Introduction

Errors handled by DUDRIVER fall into three general categories:

- A command which has timed out after being issued to a controller.
- An MSCP end message received from a controller indicating that an error occurred.
- Loss of the SCS connection with the MSCP server in a controller.

This chapter deals with the detection of such errors, and DUDRIVER's response to them. For errors related to a particular unit, only non-shadowed disks will be considered here; error detection and handling for shadow set virtual units is covered in a later chapter.

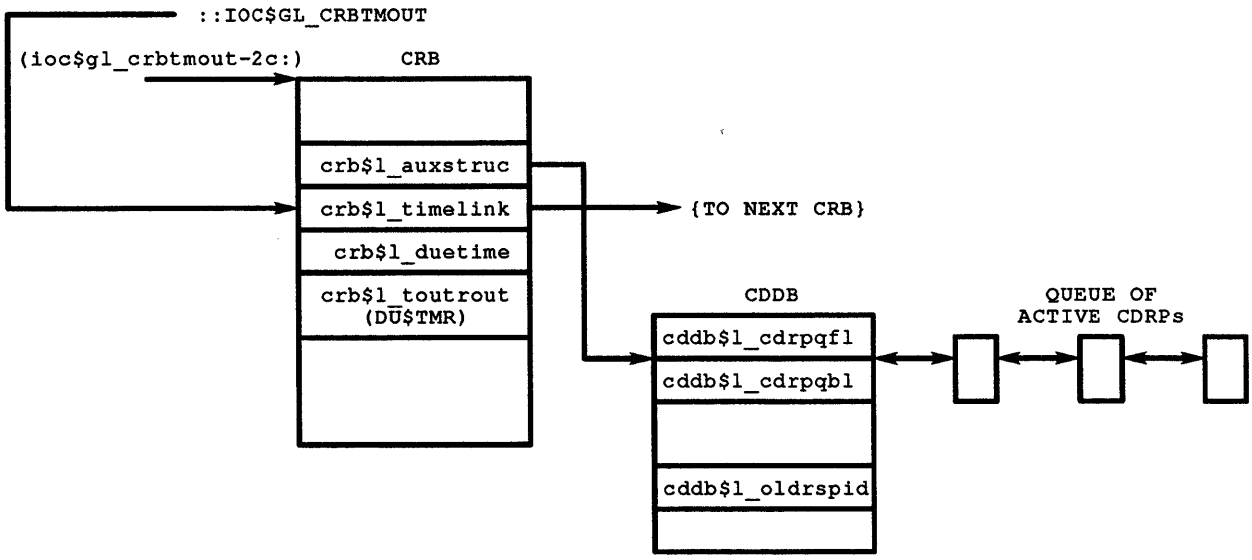
Also presented are the details of DUDRIVER's synchronizing activity with a controller's MSCP disk server, the handling of a broken SCS connection between DUDRIVER and an MSCP server, and mount verification of non-shadowed disks.

4.2 DUDRIVER Timeout Mechanism

During disk class driver controller initialization performed by routine DU_CONTROLLER_INIT, a channel request block (CRB) is initialized and inserted in the CRB timeout list, *IOC$GL_CRBTMOUT*. This CRB is setup to periodically timeout every *N* seconds, where *N* is the content of the CNTRLTMO field of the CDDB associated with the controller. DU_CONTROLLER_INIT also sets the TOUTROUT field of this CRB to contain the address of the disk class driver timeout routine, *DU$TMR*.

By means of a *Timer Queue Entry* (TQE), routine EXE$TIMEOUT in module TIMESCHDL is called once a second to perform the "once a second functions", such as checking for device and lock management request timeouts, and updating the system absolute time in seconds. One of these "once a second functions" is to scan the CRB timeout list for CRBs which have timed out. In particular, when the CRB associated with a particular DSA controller times out, routine DU$TMR in DUDRIVER is called. Figure 4–1 illustrates the relationship of the CRB and the Timeout Links:

Figure 4–1: CRB Timeout Mechanism and linkage



4.2.1 Overview of the Timeout Mechanism

DU$TMR begins by extracting the address of the CDDB associated with the controller from the AUXSTRUC field of the CRB. It then determines if any commands are currently active for this controller by checking the CDRP queue in the CDDB.

If no CDRPs are queued to the CDDB, then no commands are active for the controller. In this case, DU$TMR merely issues what is effectively a NOP *(Get Unit Status)* to the controller so that the controller does not timeout this host due to inactivity between them. After issuing the NOP, DU$TMR then branches to routine DUTU$DODAP to perform determine access paths processing.

If one or more CDRPs are queued to the CDDB, then for each such CDRP there is an active MSCP command which has been issued to the controller, but for which no corresponding end message has been received from the controller. DU$TMR examines the oldest active CDRP and determines if it was queued after the last call to DU$TMR. If so, then it is not considered to be "very old", and DU$TMR does nothing on this call other than reset the CRB's timeout.

If, however, it was queued prior to the last time DU$TMR ran, then it is considered to be "very old", and possibly "too old"; thus, DU$TMR issues a GET COMMAND STATUS to the controller for this command. If the end message for the GET COMMAND STATUS indicates that the controller has made progress on this command since it received it (or since the last GET COMMAND STATUS), then DU$TMR merely branches to DUTU$DODAP.

If the end message from the controller indicates that no progress has been made, then DU$TMR branches to *DU$RE_SYNCH_PKT* to reset the controller on the presumption that the controller is "very ill".

When issuing the NOP or a GET COMMAND STATUS, DU$TMR sets a flag to "remember" that it has done so. If the controller is unable to respond to either, then this flag will still be set the next time DU$TMR is called. This will also cause DU$TMR to branch to the code to reset the controller. The NOP (which is really a GET UNIT STATUS) and the GET COMMAND STATUS are both *immediate class commands*; they should have been responded to by the next time DU$TMR is called.

4.2.2 Detailed Flow of DU$TMR

DU$TMR Determines whether or not any CDRPs representing MSCP commands are queued to CDDB, and branches accordingly.

- Fetches address of the CDDB associated with the controller from the AUXSTRUC field of the CRB.
- Checks to verify that we still have a connection
- Tests to see if the immediate pending flag *(CDDB$V_IMPEND)* in the *CDDB$W_STATUS* field is set and branches to DU$RE_SYNCH if it is (catch timeout routine collisions)

NOTE

If the IMPEND flag is found to be set here, then an immediate command issued from this routine during the previous pass has not yet been responded to by the controller. The controller is presumed to be "very ill". DU$TMR takes no further action here, but rather branches to *DU$RE_SYNCH* to reset the controller. The error code *EMB$K_CLTRES_IMTMO* is passed to DU$RE_SYNCH.

- Checks if any CDRPs are queued to the CDDB. (Such CDRPs represent commands issued to the controller for which end messages have not yet been received.)

4.2.2.1 No Commands active for Controller

DU$TMR issues a NOP to the controller so that the controller does not timeout this host due to inactivity between them, and then invokes DAP processing for units on that controller.

- Clears OLDRSPID field in CDDB since no MSCP commands are active. (Prevents a rare "inadvertent comparison error" in the case of where commands are active.)
- Tests to see if the DAP CDRP is currently in use by testing the DABBSY bit in the status field. If it is, further tests will be performed to determine why and what action is required.
- The IMPEND flag is now set since an immediate command is about to be sent to the controller.

- Allocates the DAP CDRP by setting the DAPBSY flag in the CDDB$W_STATUS field of the CDDB to send the immediate command
- Establishes the *Credit_stall* routine as the new timeout routine and sets the crb$l_duetime (45 seconds in VMS V5.5).
- Allocates an RSPID and a Message Buffer
- Resets the normal Timeout routine (DU$TMR) and duetime in the CRB.
- Issues a GET UNIT STATUS command to controller for unit 0, even if there is no unit 0. (state 2)

NOTE

Effectively serves as a NOP so that the controller won't timeout this host due to inactivity.

Fork thread suspended here until corresponding End Message received from controller.

- When the end message corresponding to GET UNIT STATUS is received,
  - Saves *Load Available* information returned from the controller in the *CDDB$W_LOAD_AVAIL* field
  - Message buffer and RSPID recycled.
  - The IMPEND flag and DAPBSY flag are cleared.
  - This routine branches to DUTU$DODAP to perform determine access paths processing.

4.2.2.2 Commands Are Still active for Controller

If the oldest active command has been around for a long time (since the last timeout), DU$TMR interrogates the controller to see if any progress has been made on this command. It also invokes DAP processing.

- If one or more CDRPs are queued to the CDDB, then the CDRP at the head of the queue represents the oldest active command (new CDRPs are inserted at the tail of the queue). This oldest CDRP is examined to see how long it has been around.
- If this CDRP's RSPID field is different from the OLDRSPID field in the CDDB, then the oldest command was queued to the CDDB since the last call of the timeout routine. Consequently, the oldest active command is not considered to be "very old".
  - DU$TMR resets the OLDRSPID field in the CDDB to contain a copy of the content of the RSPID field from the CDRP. It also sets the OLDCMDSTS field in the CDDB to contain -1.

NOTE

If this same CDRP is found at the head of the queue on the next pass through DU$TMR, it will then be considered "very old". Then, as will be

explained later in this section, a GET COMMAND STATUS will be issued to the same controller to which the command was sent. The command status returned in the corresponding end message will be compared against the OLDCMDSTS to determine if the controller has made any progress on the command.

- — If the controller type indicates an HSC, proceed to perform a Get Unit Status as for an empty CDRP queue
- — If Load information is specifically requested (as indicated by bit MSCP$V_CF_LOAD in the CDDB$W_CNTRLFLGS field) then perform a Get Unit Status as for an empty CDRP queue.
- — DU$TMR resets the CRB$L_DUETIME field to contain the current time plus the "controller delta" stored in the CDDB$W_CNTRLTMO field.
- — Clears the IMPEND flag to indicate no immediate commands are pending
- — Branchs to DUTU$DODAP to perform determine access paths processing.

• If this CDRP's RSPID field matches the OLDRSPID field in the CDDB, then the oldest active command has been queued to the CDDB for at least one full controller timeout period and is considered to be "very old" (perhaps "too old").

- — Tests to see if the DAP CDRP is currently in use by testing the DABBSY bit in the status field. If it is, further tests will be performed to determine why and what action is required.
- — Allocates the DAP CDRP by setting the DAPBSY flag in the CDDB$W_STATUS field of the CDDB to send the immediate command
- — Sets the IMPEND bit to indicate that an immediate mode command is about to be issued (Get Command Status)
- — Establishes the Credit_stall routine as the new timeout routine and sets the crb$l_ duetime (45 seconds in VMS V5.5).
- — Allocates an RSPID and a Message Buffer
- — Resets the normal Timeout routine (DU$TMR) and duetime in the CRB.
- — Issues GET COMMAND STATUS command to the controller to ask if the controller has made any progress on this command since it was sent to the controller (or since the controller received the last GET COMMAND STATUS inquiring about this command.)
  - o If the UCB actually represents a shadow set virtual unit (MSCP$V_SHADOW flag is set in the MSCPUNIT field of the UCB), or if the oldest active command's function code in the CDRP$W_FUNC field is IO$_CRESHAD, then routine *DU$SHADOW_GTCMD_UNIT* is called to fetch the shadow set virtual unit number.
  - o The command reference number inserted into the GET COMMAND STATUS packet is fetched from the OLDRSPID field of the CDDB.
- — Issues the Get Command Status to the controller

NOTE

The fork thread is suspended here until the corresponding end message received from controller.

— When the end message corresponding to GET COMMAND STATUS is received from the controller, a 32-bit unsigned comparison of the CMD_STS field of the end message with the OLDCMDSTS field in the CDDB is made.

If the CMD_STS is smaller, then progress has been made by controller on the command in question:

- o CMD_STS field in message copied to OLDCMDSTS field in CDDB.
- o Buffer containing end message and RSPID both recycled.
- o The IMPEND flag and DAPBSY flag are cleared.
- o If the controller type indicates an HSC or if the MSCP$V_CF_LOAD flag is set indicating Load Availability information is required, then branch back to perform a Get Unit Status
- o Branch made to DUTU$DODAP to perform "determine access paths" processing.

If the CMD_STS is not smaller, then no progress has been made by the controller since it received the command, or since the last GET COMMAND STATUS. The controller is presumed to be "very ill", and a branch is made to DU$RE_SYNCH_PKT to log an EMB$K_CTLRES_TMO error due to no progress being made on the MSCP command. The RE_SYNCH_PKT routine will also reset the controller.

4.3 MSCP End Messages With Error Status Codes

DUDRIVER's involvement with a file read or write request is triggered by its start I/O routine, DU_STARTIO, being handed an IRP. This IRP represents a single transfer segment, that is, a set of consecutive virtual blocks which map to a set of consecutive logical blocks. Due to file fragmentation, it may be necessary for DUDRIVER to transfer several segments in order to satisfy one $QIO request.

For each transfer segment, DUDRIVER builds and sends an MSCP Command to the primary path controller for the disk. (Where there is only one controller for the disk, by default that controller is the primary path controller.) The data is then exchanged between VAX memory and the controller when the controller is ready.

Upon completion of the exchange, the controller sends an *MSCP End Message* to DUDRIVER. The End Message provides the disk class driver with a completion status code for the segment. Based on this status code, DUDRIVER will decide whether to enter an error handling routine, or to continue with normal processing of the $QIO request.

Assuming that the status code indicates "SUCCESS", then normal processing would be to enter I/O postprocessing. There, a determination is made to see if more transfer segments must be exchanged with the controller to complete the request. If so, then I/O postprocessing updates the IRP to reflect the next segment, and passes the updated IRP to DUDRIVER's start I/O routine. If the entire request has been satisfied, then I/O postprocessing passes the IRP to the I/O completion routines for AST delivery and event flag posting.

Figure 4–3 and Figure 4–4 illustrate the flow of MSCP **Commands**, **Data**, and **End Messages** for a read request and a write request. Except for a couple of "request data" messages, port level protocol has been left out of the diagrams since it contributes nothing to the discussion at hand.

In both the read request and write request, eight blocks are to be transferred. Due to file fragmentation, both requests must be broken up into two segments. Each segment consists of four blocks. The important point to notice in both of these diagrams is that an MSCP Command and matching End Message are associated with each segment of the request.

It is NOT the case that there is a single MSCP Command and End Message for the entire request unless the request can be mapped to a single transfer segment.

The status code returned in an MSCP End Message is 16 bits wide and consists of two fields. The low order 5 bits constitute a *Major Status Code*, and the high order 11 bits are called a *Sub-Code*.

The major status code conveys the normal information needed by class drivers; therefore, all controllers must return the same major status codes for similar situations.

Sub-codes exist for unusual situations, and to refine a major status code. They are primarily used for diagnostic purposes, and should not generally be needed by a host. Unlike VMS condition values, MSCP status codes indicate success if the *low bit is a "0"*. In fact, the major status code for success is 0. Figure 4–2 depicts the format of the 16-bit status code returned in an MSCP End Message:

Figure 4–2: MSCP End Message Status Return Format

Figure 4–3: MSCP Read Request Message Flow

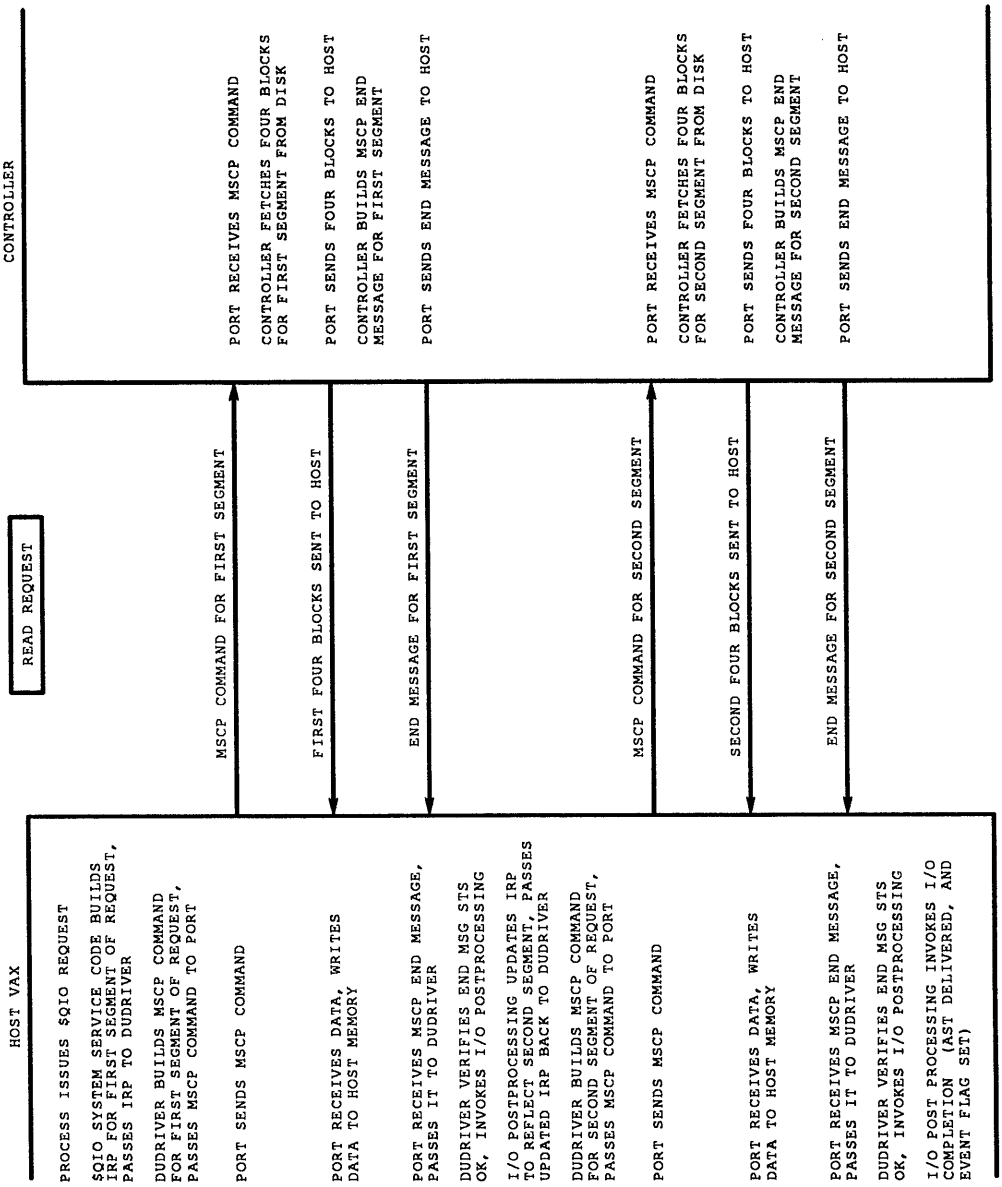
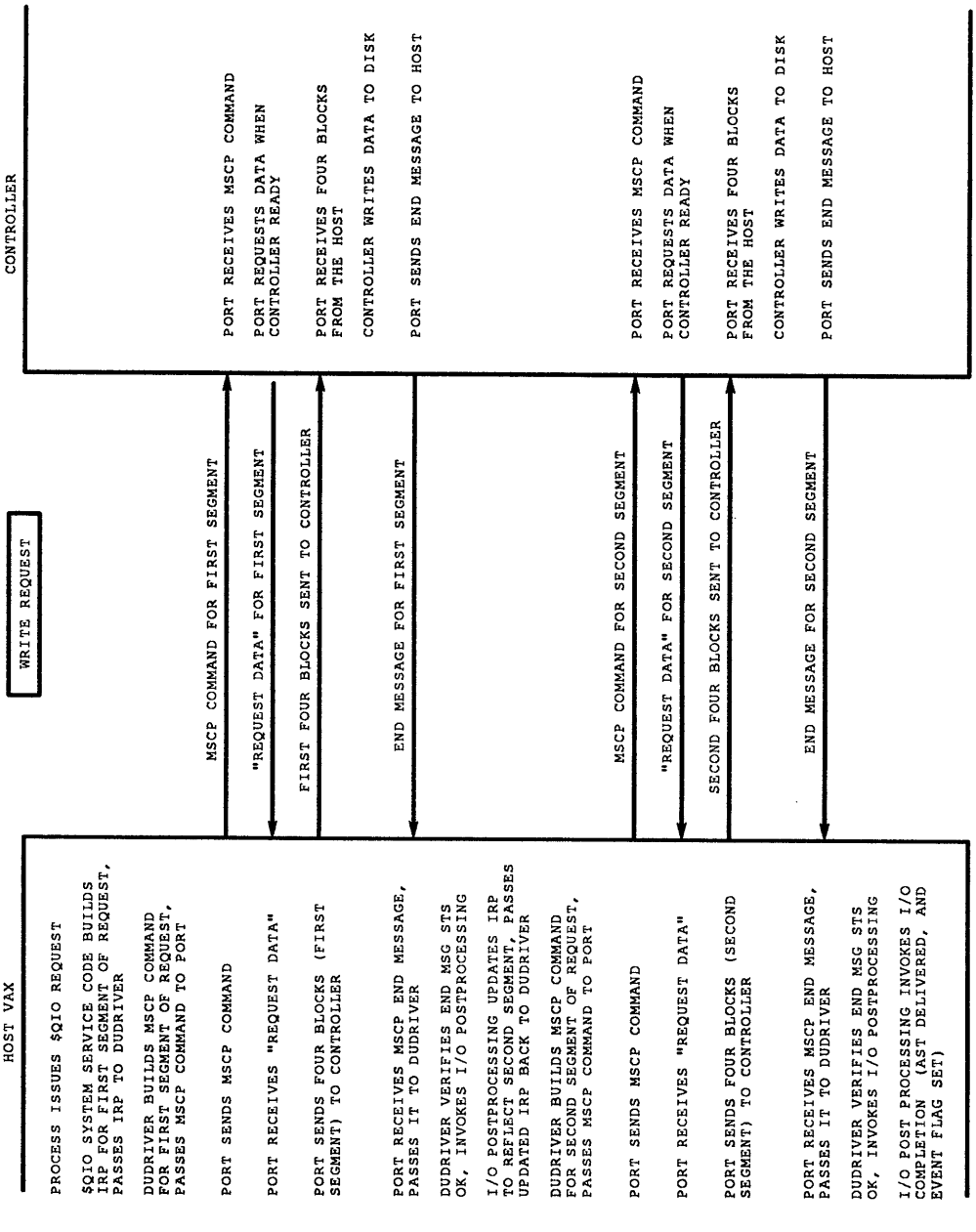


Figure 4–4: MSCP Write Request Message Flow



4.3.1 Detecting File Read/Write Errors and Dispatch

When DUDRIVER's start I/O routine, DU_STARTIO, is passed an IRP representing a file read or write transfer segment, it allocates a RSPID and associated RDT entry. The address of the CDRP attached to the IRP is stored in the RDT entry, and the RSPID is stored in the CDRP. It also allocates an SCS message buffer in which to build an MSCP Command to be sent to the controller. The RSPID is copied into the message buffer, and the address of the message buffer is saved in the CDRP.

Some of the fields in the CDRP are filled in, and then DU_STARTIO branches either to START_WRITEPBLK or START_READPBLK. There, data is supplied for the remaining fields of the CDRP, the MSCP Command is built, and the CDRP (containing the address of the MSCP Command buffer) is passed to the SCS layer to effect transmission of the MSCP Command to the controller.

Passing the CDRP to the SCS layer is done by the macro *SEND_MSCP_MSG*. This macro calls routine FPC$SNDCNTMSG in the SCS layer of the appropriate port driver (PADRIVER, PIDRIVER, PEDRIVER, or PUDRIVER). SCS and PPD routing information is added to the buffer containing the MSCP Command, and the buffer is then passed to the port for transmission.

The request is then suspended, with its context saved in the CDRP. Part of this context is the PC at which to resume when the End Message corresponding to the MSCP Command is received. This will be the address of the instruction following the SEND_MSCP_MSG macro.

The controller and port(s) exchange the data in the transfer segment to be read from or written to the disk. The controller then releases an End Message to the host containing both the 16-bit MSCP status code and a copy of the RSPID in the MSCP Command. The SCS layer of the host's port driver routes the End Message to DUDRIVER's message input dispatching routine, *DU$IDR*.

DU$IDR fetches the address of the CDRP contained in the RDT entry identified by the RSPID. It then resumes the request represented by the CDRP at the address in the "saved PC" field of the CDRP.

The request resumes immediately after the SEND_MSCP_MSG macro. Here, the *IF_MSCP* macro is used to test the major status code returned in the End Message. If any of the low order five bits of the MSCP$W_STATUS are nonzero, the controller is indicating to the host that something went wrong with this transfer segment. A branch is taken to routine *TRANSFER_MSCP_ERROR* to determine how to handle the error.

At routine TRANSFER_MSCP_ERROR is a macro called, *DO_ACTION*, followed by an "Action Table". For each possible 5-bit major status code returned in the End Message, the action table specifies a routine to which a dispatch is to be made. The instructions generated by the DO_ACTION macro cause the dispatch to actually happen.

NOTE

The instructions generated by the DO_ACTION macro varies with the values specified for its parameters. In this particular case, it generates a call to a subroutine, DUTU$INTR_ACTION_XFER, which uses the stack in a "rather crafty" fashion.

The subroutine causes the dispatch to occur as if it were a "branch" directly from location TRANSFER_MSCP_ERROR instead of a call through this intermediate subroutine. The reader is referred to module *DUTUMAC* for the definition of this macro as well as others used in DUDRIVER.

4.3.2 Errors Returned in End Messages for File Read/Write Requests

The first thing TRANSFER_MSCP_ERROR does is select a VMS condition value to correspond to each of the possible MSCP major status error codes. Table 4–1 lists the major status codes, the corresponding VMS condition values and descriptions of each.

Table 4–1: MSCP to VMS Error Code mapping

(hex)-MSCP Major Status Code	(hex)-VMS Condition Code	Description
(00)-MSCP$K_ST_SUCC	(0001)-SS$_NORMAL	Normal successful completion
(04)-MSCP$K_ST_AVLBL	(01AC)-SS$_MEDOFL	Unit identified by unit number field in End Message is in the "unit available" state.
	(008C)-SS$_DRVERR	If drive inoperative
(03)-MSCP$K_ST_OFFLN	(01A4)-SS$_MEDOFL	Unit identified by unit number field in End Message is in the "unit offline" state. There are several reasons that make this possible: unit unknown or online to another controller, no volume mounted, drive disabled by RUN/STOP switch, unit disabled by internal diagnostic, etc.
(08)-MSCP$K_ST_DATA	(2144)-SS$_FORCEDERROR	Transfer data error. Invalid or uncorrectable data was obtained from a drive. Typical causes are valid header not found, data sync timeout, one through eight symbol ECC errors, and the "forced error" condition.
	(2144)-SS$_FORCEDERROR	if "forced error" condition
	(01F4)-SS$_PARITY	If not forced error condition
(0B)-MSCP$K_ST_DRIVE	(008C)-SS$_DRVERR	The controller has discovered an error within the drive. The error is typically, but not always, mechanical in nature, since most non-mechanical errors are reported as "data errors".
(0A)-MSCP$K_ST_CNTRL	(0054)-SS$_CTRLERR	The controller has encountered an internal error.

Table 4–1 (Cont.): MSCP to VMS Error Code mapping

(hex)-MSCP Major Status Code	(hex)-VMS Condition Code	Description
(06)-MSCP$K_ST_WRTPR	(025C)-SS$_WRITLCK	Command required that data be written to a write protected unit.
(07)-MSCP$K_ST_COMP	(005C)-SS$_DATACHECK	COMPARE HOST DATA command, read compare operation, or write compare operation found differences in the data on the unit and in the host buffer; or COMPARE CONTROLLER DATA command found different data on different members of a shadow set.
(05)-MSCP$K_ST_MFMTE	(00BC)-SS$_FORMAT	Volume mounted on unit appears to be formatted incorrectly.
(02)-MSCP$K_ST_ABRTD	(002C)-SS$_ABORT	Command aborted by ABORT command.
(01)-MSCP$K_ST_ICMD	(0054)-SS$_CTRLERR	Invalid command. A controller returns this status code because it believes the host made an error in one of two ways: • Host supplied invalid parameter values in an MSCP Command (e.g. nonexistent logical block number, ...). • Protocol error in MSCP Command controller received from host (e.g. reserved field does not contain proper quantity, command too short to contain all the required parameters, ...).
(09)-MSCP$K_ST_HSTBF	(034C)-SS$_IVBUFLEN	Host buffer access error. The controller encountered an error while trying to access a buffer in host memory. This error is also returned when an MSCP Command's buffer descriptor or byte count violate any communications mechanism dependent restrictions. It is not, however, used to report errors encountered by the port(s) when transferring packets between the host and the controller. Those are handled by terminating the connection between the class driver in the host and the server in the controller. Two typical causes would be a local DSA controller getting a nonexistent memory error or host memory parity error.

Table 4–1 (Cont.): MSCP to VMS Error Code mapping

(hex)-MSCP Major Status Code	(hex)-VMS Condition Code	Description
(0C)-MSCP$K_ST_SHST	(2284)-SS$_SHACHASTA	Shadow set state change. Member must be removed from shadow set because host requested removal or member is no longer operative.

4.3.3 Handling Errors Returned in Read/Write End Messages

Once routine TRANSFER_MSCP_ERROR has selected the appropriate VMS condition value, it executes common steps for handling all errors except the "Invalid Command", the "Host Buffer Access Error" and the "Available" error with a subcode of "Inoperative". The handling of these special cases can be found in Section 4.3.3.1.

For the normal error path, the following is performed:

- First, a branch is taken to routine *TRANSFER_RTN_BCNT*. At this routine, the actual number of bytes transferred is extracted from the BYTE_CNT field of the End Message, and combined with the VMS condition value to form a quadword in the proper format for an I/O status block.
- If the controller reports a "bad block" in the FLAGS field of the End Message, then a branch is taken to *XFER_REPLACE* to consider performing host initiated bad block replacement.

NOTE

If this is the case and a branch to XFER_REPLACE is made, then the flow of handling the error does not return here, and no further processing will be done by these steps. This, however, should happen only with controllers which do not perform their own *Bad Block Replacement* (BBR) , namely most local controllers. This should not happen with controllers which, from local VMS's point of view, handle their own BBR (i.e. HSCs, ISEs, KDM70s and remote VAXes running the MSCP server).

- If the controller does not report a "bad block", then this flow continues by branching to routine FUNCTION_EXIT where the following occurs:
  - — If the translated VMS status code indicates that an error condition exists (low bit clear) and this is not the mount verification IRP, then a branch subroutine is taken to routine *DU$MSG_ERR_HNDLR*[1] to determine whether the devices error count should be incremented and/or whether to log an error. See Section 4.3.5 for a description of routine DU$MSG_ERR_HNDLR.
  - — The error logging in progress bit *CDRP$M_ERLIP* is cleared

[1] This routine was introduced in VMS V5.4-3 to identify the correct conditions under which to increment a device's error count and when to log the error

- — SCS resources held by this CDRP are released.
- — FUNCTION_EXIT branches to IOC$ALTREQCOM to request I/O completion activity.

• IOC$ALTREQCOM performs the following tasks:

- — VMS condition values corresponding to errors have the low order bit clear. IOC$ALTREQCOM tests for this, and, finding it to be the case, calls *EXE$MOUNTVER* to force mount verification to be performed for the unit.

  Mount verification is covered later in this chapter. It is important however to realize that this procedure may result in the transfer segment being successfully retried. It is possible for mount verification to change the VMS condition value to "success" (SS$_ NORMAL).

- — Once mount verification completes (or is terminated), the final I/O status block quadword is stored in the MEDIA field of the IRP. IOC$ALTREQCOM then passes the IRP to I/O post processing.

  If the VMS condition value that emerges from mount verification indicates other than "success", the request will be terminated without attempting further segments, and the error will be stored in the I/O status block specified by the process which issued the $QIO request. If the condition value does indicate "success", then the request continues on as if the error had never occurred. The problem should be transparent to the process, other than of course, the delay due to mount verification.

4.3.3.1 Specially Handled Error Conditions

4.3.3.1.1 *Invalid Command* Major Status Code

For most cases of the "Invalid Command", DUDRIVER is given a second chance to "get it right". For LBN and BYTE_CNT subcodes or Invalid MSCP Modifier errors, control is transferred to FUNCTION_EXIT. For all other errors, control is passed to routine *DU_BEGIN_IVCMD* where an Invalid Command Sequence is set, an Errorlog entry is made and the function code and modifiers are extracted from the CDRP and are used to dispatch to the appropriate routine (ie: START_READPBLK). Each routine will test to see if IVCMD handling is in progress *(macro IF_IVCMD)* and will transfer control to routine *TRANSFER_IVCMD_END* if appropriate.

TRANSFER_IVCMD_END will deallocate all SCS resources held by the CDRP, and then resubmit the request to DUDRIVER. This resubmission is not to the ordinary start I/O entry point, but rather to an alternate "restart I/O" entry point called *DU_RESTARTIO*.

DU_RESTARTIO begins directly with the step of copying the address of the CDT from the UCB into the CDRP. It then executes exactly the same steps that DU_STARTIO would from that point on.

If this second attempt also results in an End Message with a major status code of MSCP$K_ ST_ICMD, then the VMS condition value SS$_CTRLERR is selected to form an I/O status block quadword with a byte count of 0. Routine FUNCTION_EXIT is entered, and the flow proceeds as in the previous cases.

4.3.3.1.2 *Host Buffer Access Error* Major Status Code

There are two possibilities for a "host buffer access error":

- If the sub-code indicates an *Odd Byte Count* error caused the problem, then error handling branches to TRANSFER_RTN_BCNT and proceeds as already described above.
- For any other subcode, a branch is taken to INVALID_STS where routine DU$RE_SYNCH is called to reset the controller. (Details of resetting a controller are presented later in this chapter.)

4.3.3.1.3 *Available* Major Status Code

For the Available status code, the VMS condition code *SS$_DRVERR* is placed in the high order word of R0 if the subcode indicates that the device is inoperative. For all other subcodes the VMS condition value is left as SS$_MEDOFL. This test occurs at label *TRANSFER_MEDOFL*. Control is then transferred to routine TRANSFER_RTN_BCNT.

4.3.3.1.4 All Other Errors

All values for the major status code in an End Message other than what is listed in Table 4–1 are considered "unexpected". If one is received, the controller is presumed to be "very ill"; so the code branches to DU$RE_SYNCH to reset the controller.

4.3.4 Errors Returned in Other End Messages

There are three other situations wherein DUDRIVER checks for an invalid major status code returned in an End Message. These three situations arise when issuing an

- *IO$_NOP* for a unit.

  This function is turned into a SET UNIT STATUS, using current status, by routine *START_NOP*.
- *IO$_PACKACK* for a unit.

  Routine *START_PACKACK* performs this function by issuing an ONLINE followed by a GET UNIT STATUS for the unit. It is typically done when a unit is first discovered on an "MSCP speaking" controller, or when trying to establish a path to the unit during mount verification.
- *IO$_AVAILABLE* for a unit.

  The IO$_AVAILABLE function causes an AVAILABLE command to be issued to the controller for a unit. This is typically done as part of dismounting the unit, and can involve optionally spinning down the volume

The following is a table of valid major status error codes which may be returned by a DSA controller for each of these situations. These codes are a subset of those already listed for file read/write End Messages.

IO$_NOP	IO$_PACKACK	IO$_AVAILABLE
MSCP$K_ST_OFFLN	MSCP$K_ST_OFFLN	MSCP$K_ST_OFFLN
MSCP$K_ST_AVLBL	MSCP$K_ST_AVLBL	MSCP$K_ST_AVLBL
MSCP$K_ST_CNTLR	MSCP$K_ST_CNTLR	MSCP$K_ST_CNTLR
MSCP$K_ST_DRIVE	MSCP$K_ST_DRIVE	MSCP$K_ST_DRIVE
MSCP$K_ST_SHST	MSCP$K_ST_SHST	MSCP$K_ST_SHST
MSCP$K_ST_ICMD	MSCP$K_ST_ICMD	MSCP$K_ST_ICMD
	MSCP$K_ST_ABRTD	MSCP$K_ST_ABRTD
		MSCP$K_ST_MFMTE
		MSCP$K_ST_DATA

Any other major status error codes are considered as "unexpected". The controller is presumed to be "very ill" if one is received; so a branch is made to DU$RE_SYNCH to reset the controller.

4.3.5 Error Logging and Error Count Incrementing

Error logging and device error count incrementing is handled by routine DU$MSG_ERR_HNDLR. It determines under what conditions errors are to be logged to the Errorlog file *(SYS$ERRORLOG:ERRLOG.SYS)* and under what conditions the device error count *(UCB$W_ERRCNT)* is to be incremented. It is called when an error condition is detected in an MSCP End Message from routine FUNCTION_EXIT.

The general flow through the routine is as follows:

- Test bit *MSCP$V_EF_ERLOG* to determine if the Errorlog entry was expected. If so, the device error count will not be incremented, but an Errorlog entry will be recorded (by calling routine *ERL$LOGSTATUS*). Control is then returned to FUNCTION_EXIT to continue processing.
- Determine if error logging is already in progress (bit *CDRP$V_ERLIP*. If so, the device error count will not be incremented, but an Errorlog entry will be recorded (by calling routine *ERL$LOGSTATUS*). Control is then returned to FUNCTION_EXIT to continue processing.
- Determine if the device error count is really to be incremented and/or if an Errorlog entry is to be made:
  - — If the MSCP major status code indicates an Invalid Command (ICMD) the VMS condition value is set to SS$_CTRLERR and a return is made without incrementing the device error count or logging the error.
  - — If the MSCP major status code is Available (AVLBL), a test is made to see if the actual error was due to the device being inoperative. If so, the VMS condition value will have been set to SS$_DRVERR as described in Section 4.3.3.1.3 and an Errorlog entry will be logged as well as the device error count will be incremented.
  - — If the MSCP major status code is Offline (OFFLN) or it is the Available code with a subcode other than inoperative, only an Errorlog entry will be logged.

  - If the MSCP major status code indicates a Controller Error (CNTLR), a Forced Error (DATA) or a Format Error (MFMTE) and this is not a bad block replacement, both the device error count will be incremented and an Errorlog entry will be logged. For bad block replacement, only the Errorlog entry will be generated.
  - For Drive Errors (DRIVE), both the device error count will be incremented and an Errorlog entry will be created.
- All other errors are ignored by this routine and control is passed back to routine FUNCTION_EXIT.

4.4 Synchronizing with an "MSCP Speaking" Controller

Anytime an SCS connection is established between the local disk class driver and an MSCP disk server, it is necessary to synchronize the activity between them. This is done by forcing the dialogue between the driver and the server into a known state. In doing so, we guarantee that there are no outstanding MSCP commands from the server's point of view.

The MSCP server has the responsibility for insuring that this guarantee is met. To this end, before allowing synchronization to complete, the server

- Terminates any outstanding MSCP commands it has received.
- Does not send end messages corresponding to the terminated MSCP commands to the host
  .

Thus, once the SCS connection is established, the disk class driver can reissue outstanding MSCP commands, as well as issue new ones, without worrying about side effects such as duplicate command reference numbers.

A disk class driver must synchronize with an MSCP disk server whenever the host boots or recovers from a power failure, whenever the SCS connection between the two is broken, or as part of the recovery mechanism when certain types of errors occur.

4.4.1 Errors Causing Resynchronization with an MSCP Server

There are six general situations which cause DUDRIVER to resynchronize with a DSA controller's MSCP server:

- The controller has made no progress for "too long" a time on the oldest active command issued to it by the local host.
- An immediate class command, either GET UNIT STATUS or GET COMMAND STATUS, issued to the controller by the local host has timed out.
- The local host has received an invalid attention message from the controller.

  This determination is made based on the MSCP$B_OPCODE field of the attention message. Valid attention message op codes are:

Code	Meaning
AVATN	Unit Available Attention
DUPUN	Duplicate Unit Attention
ACPTH	Access Path Attention

If the op code field of an attention message indicates anything else, the attention message is considered invalid.

- The local host has received an end message containing an invalid MSCP status code.
- DUDRIVER fails to allocate an SCS message buffer in which to build the *SET CONTROLLER CHARACTERISTICS* command after establishing an SCS connection with the server. (e.g. Insufficient nonpaged pool could cause this.)
- The SCS connection between DUDRIVER and the server is unexpectedly broken.

  Some of the typical causes of this are
  - — The controller hangs.
  - — The controller experiences a power failure.
  - — An SCS protocol error.
  - — If the controller is CI-based, both CI paths A and B go "from good to bad".
  - — If the controller is CI-based, a CI port error (local or remote) causing port reinitialization.

As usual, unless otherwise noted, the term "controller" refers to a local "MSCP speaking" controller, a remote "MSCP speaking" controller, or a remote VAX emulating an "MSCP speaking" controller by running the VMS based MSCP server.

4.4.2 Overview of Resynchronization Due to Errors

There are two major routines involved in handling the resynchronization that occurs between the disk class driver and an MSCP server.

DU$CONNECT_ERR is invoked by the SCS layer when the SCS connection between the driver and the server is unexpectedly lost.

DU$RE_SYNCH (or DU$RE_SYNCH_PKT, an alternate entry point) is invoked by:

- DUDRIVER's timeout mechanism, DU$TMR, to handle the "no progress on oldest command" and "timed out immediate command" situations.
- DUDRIVER's attention message handler, ATTN_MSG, when it receives an invalid attention message.
- DUDRIVER's invalid MSCP status handler, INVALID_STS (also called DU$INVALID_STS), which handles end messages received with "unexpected" MSCP status codes.
- DUDRIVER's routine, MAKE_CONNECTION, establishing an SCS connection with an MSCP server, when it fails to allocate an SCS message buffer in which to build a SET CONTROLLER CHARACTERISTICS command.

Most of what each of these routines does is common to both. Therefore, DU$RE_SYNCH and DU$CONNECT_ERR are, in fact, alternate entry points to the same set of instructions. Here is a brief summary of the principal steps performed by both routines, accompanied with a flowchart; detailed analysis follows in the next section.

DU$RE_SYNCH begins by first determining if the controller is actually a VAX running the VMS based MSCP server by examining *MSPC$K_CM_EMULA* bit in the CNTRLMDL field of the CDDB for the controller. If the controller is not a VAX, then it sets the RESYNCH flag in the CDDB's STATUS field; this will cause the local host to later reset the controller by issuing a *Host Clear* operation to it. If the controller is really a remote VAX, the local host should not attempt to cause the remote VAX to reload, but merely break the SCS connection with its MSCP server; so the RESYNCH flag is left clear.

At this point, the flow of DU$RE_SYNCH merges in with the beginning of routine DU$CONNECT_ERR.

Since the SCS connection with the controller's MSCP disk server is either already broken, or about to be, both DU$RE_SYNCH and DU$CONNECT_ERR must stall all new I/O requests being handed to DUDRIVER's start I/O routine. This is done by incrementing the RWAITCNT field of every UCB linked to the controller's CDDB. Until the RWAITCNT field of a UCB is restored to 0, any new IRPs for the unit associated with that UCB will merely be inserted by the start I/O routine into the UCB's pending IRP queue (UCB$L_IOQFL).

Also, I/O requests that have already progressed past the start I/O routine's RWAITCNT checkpoint must be gathered up for resubmission to DUDRIVER in the event that error recovery being triggered by DU$RE_SYNCH or DU$CONNECT_ERR is successful. CDRPs representing these "active" requests are collected and inserted into the restart queue on the CDDB in the exact order in which they were originally handled by DUDRIVER.

At this point, consider only routine DU$RE_SYNCH, and not DU$CONNECT_ERR. Now is when the RESYNCH flag is used. If this flag is set, then a "host clear" is done to the controller to force it to reset itself. This is accomplished by issuing to the controller an *MSCP RESET* followed immediately by an *MSCP START*.

Then DU$RE_SYNCH merely returns to its caller. As a result of the controller being re-set, the SCS connection with the controller's server is broken; and this in turn will cause DU$CONNECT_ERR to be invoked. Of course, DU$CONNECT_ERR need not increment the RWAITCNT field since that was already done by DU$RE_SYNCH. However, it will go through the formality of trying to gather up "active" CDRPs; but this formality will happen quickly since that was also already done by DU$RE_SYNCH.

If the RESYNCH flag is clear, then one of the following two situations is true:

- This is currently DU$RE_SYNCH executing and the controller is a VAX emulating an "MSCP speaking" controller. Given this to be true, the next step is to break the SCS connection with the VAX's MSCP server by means of the SCS service DISCONNECT.
  - — If the *CDDB$V_PATHMOVE* bit in the CDDB$W_STATUS field indicates that this is a pathmove, the *SCS$C_USE_ALTERNATE_PORT* reason code is passed to the disconnect service.
  - — If the CDDB$V_PATHMOVE bit is clear, then a normal disconnect reason is passed.

- This is currently DU$CONNECT_ERR executing, either because the SCS connection with a controller's server was unexpectedly lost, or it was just intentionally broken by DU$RE_SYNCH. If this situation applies, then the formality of doing an SCS DISCONNECT is still done, but only to "clean up" the local host's data structures associated with the broken connection.

Next all mapping resources owned by CDRPs on the restart queue are deallocated. This is performed by calling routine DUTU$DEALLOC_ALL for each CDRP on the restart queue.

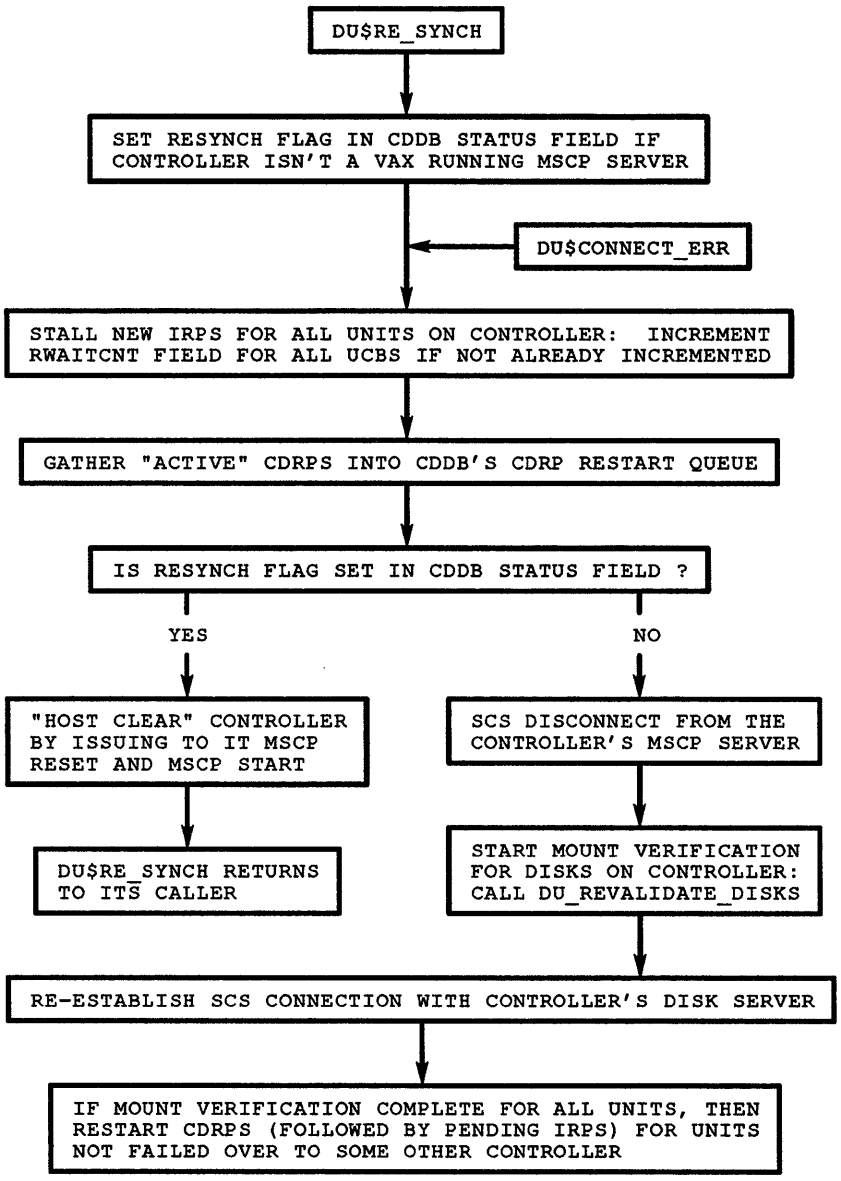
Next, *Mount Verification* is started for all disks on the controller by branching to routine *DUTU$REVALIDATE*. This may cause one or more disk units to failover to an alternate controller. Details of mount verification and disk failover are presented later in this chapter.

Once mount verification is started, the disk class driver attempts to form an SCS connection with the MSCP disk server in the controller. This is performed by calling routine *Make Connection*. If an SCS CONNECT attempt fails, the code will pause for CONNECT_DELTA seconds (10 seconds for VMS V5.5), and then retry. This will give the controller time to reload, if necessary.

Once the connection is established, a standard SET CONTROLLER CHARACTERISTICS is performed.

Finally, if mount verification is complete for all units remaining on the controller, then CDRPs in the CDDB's restart queue are retried; and then the IRPs in each UCB's pending IRP queue are unstalled. If mount verification is not finished for all units on the controller, then restarting CDRPs and unstalling pending IRPs is handled by mount verification. This basic flow is illustrated in Figure 4–5.

Figure 4–5: DUDRIVER resynchronization flow



CXN-0004-09

4.4.3 DU$RE_SYNCH and DU$CONNECT_ERR Detail

Except for the first step, this code is common to both resynchronizing with a controller and handling the loss of an SCS connection with the MSCP disk server in that controller.

- Steps that are unique to DU$RE_SYNCH.

  If the "controller" in question is actually a VAX running the VMS based MSCP server, then the local disk class driver should merely break its SCS connection with the server and not attempt a controller reset. Therefore, the content of the CNTRLMDL field of the CDDB is examined:

  — If it does not contain the value of the symbol MSCP$K_CM_EMULA, then the controller is a "real controller", and not a VAX emulating a controller. Set the RESYNCH flag in the CDDB$W_STATUS field, insuring that the controller will be reset later in this routine.

  — If the controller is actually a VAX, then the RESYNCH flag is not set.

  NOTE

  This step is executed only if DUDRIVER calls routine DU$RE_SYNCH. It is not executed by DU$CONNECT_ERR in response to an abruptly failed SCS connection. Consequently, in the case of a failed connection, the CDDB$W_STATUS will not have its RESYNCH flag set, regardless of what type of controller is involved. This fact will shortly become critical to the logical flow of events governed by this procedure.

- This step is executed only if DUDRIVER calls routine DU$CONNECT_ERR. The disconnect reason code is checked against the value SCS$C_USE_ALTERNATE_PORT. If it is equal, a Path move is indicated by setting the CDDB$V_PATHMOVE bit in the CDDB$W_STATUS word.
- Next is the first step that is common to both DU$CONNECT_ERR and DU$RE_SYNCH.

  The RECONNECT and NOCONN flags in the CDDB$W_STATUS field are set to indicate that there is currently no SCS connection with the MSCP disk server in this controller, and that a reconnect attempt is in progress.

  The IMPEND, INITING, and RSTRTWAIT flags in the CDDB are cleared, indicating that:

  — There are no immediate class commands active for this controller.

  — Controller initialization is not in progress.

  — There are no CDRPs currently waiting for restart.

  Even if any of these conditions is true, they won't be much longer because of what this routine is about to do.
- Since the SCS connection with the MSCP server in the controller is about to be broken (if it isn't already), there is no need for DUDRIVER's timeout mechanism to be active for this controller. Thus, the DUETIME field of the CRB for this controller is set to minus one indicating an infinite timeout period.
- Either the connection with the controller's disk server has already been broken, or it is about to be. Therefore it is necessary to stall further I/O requests coming from the $QIO system service for any disk units on the controller.

The list of UCBs linked to the CDDB via the CDDB$L_UCBCHAIN field is scanned. The MSCP_WAITBMP bit in each UCB is checked to see if the UCB's *RWAITCNT* field has been "bumped" (i.e. is nonzero) for some other reason. If this flag is found to be set, then I/O is already stalled. If this flag is found not to be set, then the flag is set and the RWAITCNT word in the UCB is incremented, thereby stalling new I/O requests.

When DUDRIVER's start I/O routine is handed a new request, it will insert the IRP representing the request into the UCB's pending IRP queue (*UCB$L_IOQFL*). No further processing will be done by DUDRIVER for such IRPs until I/O is "unstalled" for the unit.

- Any RSPIDs and SCS message buffers held by the CDDB's permanent or DAP CDRPs are released. They won't be needed since what's happening effectively terminates the need for any outstanding GET COMMAND STATUS or DAP processing operations. Also, because of this, these CDRPs are removed from any resource wait queues they may be on.
- Next, all "active" CDRPs for disks on this controller are gathered up and placed on the CDDB's I/O restart queue, CDDB$L_RSTRTQFL.

  These CDRPs are searched for in different places:

  — First, the *Host Initiated Replacement Table* (HIRT) wait queue is checked. A CDRP for the controller in question would be found there only if the controller is local (UDA50, KDB50, etc.), and only if the CDRP is for a bad block replacement operation for some disk on the controller.

  — Then the RDT resource wait queue is checked by routine *SCAN_RSPID_WAIT* for CDRPs waiting to be allocated a RSPID and associated RDT entry.

  — The third place checked is the CDDB's queue of CDRPs for which MSCP commands have been sent to the controller, but for which end messages have not yet been received. This is performed by routine *DUTU$DRAIN_CDDB_CDRPQ*.

  — And fourth, CDRPs waiting for any SCS resources (flow control, message buffers, mapping resources, ...) must be found. To do this, the entire RDT is scanned for any CDRPs whose operation affects this controller.

  As the CDRPs are inserted into the CDDB's I/O restart queue, any RSPIDs and SCS message buffers they possess are released.

  It is important to note that the only CDRPs "gathered up" in this operation are those that relate strictly to this controller and its disks. CDRPs related to other controllers and their disks are unaffected by this whole operation.

  At this point, it should also be noted that incrementing the RWAITCNT field for each unit on the controller (done two steps ago) has an important secondary effect. EXE$MOUNTVER is about to be called for each of this controller's units which is to undergo mount verification.

  When EXE$MOUNTVER begins the MSCP specific steps for mount verification, it increments the RWAITCNT field. The RWAITCNT field is again incremented when when EXE$MOUNTVER performs an IO$_PACKACK function. The RWAITCNT field is decremented when the IO$_PACKACK function is complete, and again when the MSCP specific functions come to an end. Each time RWAITCNT is decremented, a test is made to see if it has gone to 0. If so, then the I/O requests in the pending IRP queue are unstalled.

  Incrementing RWAITCNT here before calling EXE$MOUNTVER insures that RWAITCNT will not become 0 as a result of the two decrements just described. Instead, it will become 0 when a third decrement is performed after all "active" CDRPs in the controller's CDDB restart queue have been handled.

- Two important distinctions become critical:
  - — Is the controller a "real DSA controller", or merely a VAX emulating a DSA controller?
  - — Was this procedure entered by an explicit call to DU$RE_SYNCH by DUDRIVER, or through the entry point DU$CONNECT_ERR due to an abrupt failure of the connection with the controller's disk server? If DU$RE_SYNCH was called and this is a "real DSA controller", then the RESYNCH flag in the CDDB$W_STATUS field will have been set previously. In this case:
  - — DUDRIVER will force the controller to reset itself. It does this by issuing to the controller an MSCP RESET command, followed immediately by an MSCP START command.

    Any time a DSA controller receives such a "back-to-back" pair of MSCP commands from a class driver, it will reset itself. In the case of an HSC, this means a full reboot. The analogy has often been made that sending the MSCP RESET is like pointing a gun at the controller and cocking the hammer, and then sending the MSCP START is like pulling the trigger.
  - — Having issued the MSCP RESET and START, DU$RE_SYNCH will then merely return to its caller. It does not go on to execute the remaining steps described here.
  - — As a result of the controller resetting itself, the connection between its server and the disk class driver is abruptly broken. (That's reasonable since the server is now "dead"!)

    Upon discovering this, the local host's PPD and SCS layers will notify DUDRIVER of the connection's demise by invoking DUDRIVER's connection error handler, namely DU$CONNECT_ERR. DU$CONNECT_ERR will execute the remaining steps on behalf of DU$RE_SYNCH. If, instead of DU$RE_SYNCH, this is really DU$CONNECT_ ERR executing, then major step 1 above will have been skipped. Consequently, the RESYNCH flag will not have been set, and thus the following will be true:
  - — The flow will bypass the instructions for issuing the RESET and START, so the controller will not be reset. Furthermore, a return is not made to DU$CONNECT_ ERR's caller at this time; DU$CONNECT_ERR merely goes on to the next major step in the sequence described herein.
  - — DU$CONNECT_ERR may now be executing because the connection with the controller's disk server was abruptly and/or intentionally severed by DU$RE_SYNCH. It will, of course, execute once again all the major steps described above for DU$RE_ SYNCH (except major step 1). However, it won't find much to do since DU$RE_ SYNCH has already moved all the active CDRPs to the CDDB restart queue.
  - — If the connection with the controller's disk server was lost for reasons other than DU$RE_SYNCH intentionally breaking it, then the above steps must be executed by DU$CONNECT_ERR anyway.

    Even though it does lead to some redundancy, there is justification for common code between DU$RE_SYNCH and DU$CONNECT_ERR. If this is DU$RE_SYNCH executing but the "controller" is really a VAX running the VMS based MSCP server, then DU$RE_SYNCH also bypasses the sending of the RESET and START and continues on to the next step without returning to its caller. This is done because the "right thing to do" here is not to "shoot" the remote VAX, but to merely resynchronize communications between its server and the local disk class driver by disconnecting and then reconnecting.

- If the connection between DUDRIVER and the controller's disk server has been broken, then either it happened unexpectedly due to such things as a port level protocol failure, or intentionally due to DU$RE_SYNCH resetting the controller. In either case, it is now necessary for DUDRIVER to "clean up" the data structures at its end of the lost connection. This is done by the SCS DISCONNECT service. For the DU$CONNECT_ERR routine, a check is made to see if a Pathmove has been requested. If it has, an SCS$C_USE_ALTERNATE_PORT reason code is passed to the disconnect service.

  If the SCS connection with the server is not yet broken, then this is actually DU$RE_SYNCH running; and it is now time to break the connection. This is also done by the SCS DISCONNECT service.

  DISCONNECT will return to nonpaged pool the CDT representing the connection. For CI-based connections, DISCONNECT will also return to nonpaged pool the initial credit worth of buffers extended when the connection was first made, along with the special SCS receive buffer associated with this connection.
- If a CDRP reached the point of "mapping the IRP", then it owns "mapping resources" which map a transfer buffer somewhere in physical memory. These mapping resources are either a BDT entry if the controller is CI-based, or map registers if the controller is local (UDA, KDB, KDM, etc.).

  So a loop is entered which calls DUTU$DEALLOC_ALL for each CDRP in the CDDB's restart queue to release mapping resources held by that CDRP. A CDRP holds such resources if its local buffer handle address field, CDRP$L_LBUFH_AD, is nonzero.

  If the IRP is from Host Based Shadowing, it is now removed from the Restart queue and sent to post processing with a final status of SS$_DEVOFFLINE.
- All mapping resources are deallocated from the Permanent CDRP by calling routine DUTU$DEALLOC_ALL.
- Routine DUTU$REVALIDATE is called to start mount verification for each of the disks on the controller.

  This is done here to allow failover of disks which are dual-pathed. If a disk is not dual-pathed, then mount verification for the disk will wait until either the connection with the controller's server is re-established, or until the mount verification timeout period expires for that disk.
- Next DUDRIVER calls routine MAKE_CONNECTION in an attempt to re-establish the connection between itself and the controller's disk server, and to SET CONTROLLER CHARACTERISTICS.

NOTE

MAKE_CONNECTION does not return to its caller until it successfully connects to the controller's disk server. If its call to the SCS CONNECT service fails, it pauses for CONNECT_DELTA seconds, and then tries again. The reader is referred back to the chapter covering the DUDRIVER I/O DATABASE for details.

- If the Host is required for Initiating Bad Block Replacement, HIRT initialization is performed.

- The addresses of the CDT and PDT associated with the new connection are copied into all UCBs on the CDDB's UCB chain. The CDT and PDT addresses were returned by routine MAKE_CONNECTION.
- Now that there is an SCS connection with the controller's disk server, DUDRIVER's timeout mechanism is reactivated for the controller.
  - — The address of timeout routine, DU$TMR, is stored in the CRB.
  - — The DUETIME field of the CRB is set to the current time plus the content of the CNTRLTMO field of the controller's CDDB.
- DUTU$POLL_FOR_UNITS is called to poll for units that the local disk class driver may not have previously known were on this controller.
- Finally, CDRPs in the CDDB's restart queue must be restarted.

  However, mount verification may result in disks being failed over to other controllers; and it would not be "very good" to restart on this controller a CDRP for a disk that may ultimately be handled by some other controller.

  The CDDB$W_WTUCBCTR field contains the count of the number of UCBs waiting for mount verification to complete. If it is nonzero, then this routine merely returns to its caller; the end of the mount verification procedure will then restart the CDRPs. If it is zero, then this routine restarts CDRPs, but only for units still on this controller. For each unit mount verification fails over to some other controller, mount verification will remove that unit's CDRPs from the restart queue on this controller's CDDB and insert them in the restart queue on the other controller's CDDB.

  When single stream processing is complete, the RWAITCNT fields in the UCBs are decremented so as to allow new IRPs to be processed by DUDRIVER, beginning with the IRPs queued to the UCB itself.

4.5 Mount Verification

Certain serious problems may render one or more disk units inaccessible to their drivers. For example, perhaps an HSC90 crashes making all of its units inaccessible. Or perhaps only one unit on the HSC90 becomes inaccessible due to cable problems between the unit and an SDI in that HSC90. Maybe the local CI port hardware develops an internal parity error and has to be reloaded, thus making all remote disks inaccessible.

Not all such problems are caused by hardware failures. Suppose an operator accidentally spins down the wrong disk drive while users are still accessing it, and then replaces the volume in that drive with some other volume. Or perhaps the operator accidentally "popped out" the port buttons on a drive.

Sometimes the inaccessibility of a disk drive is even planned, and not a real problem at all. Consider the case of where an operator intentionally fails over a drive from one HSC to another. This would briefly make the the drive inaccessible to all hosts until they were able to acquire a new path to it through the other HSC.

Any time such a situation arises, it is desirable to quickly restore access to the unit and its volume, and do so in a manner which has minimal impact on the users of that volume. VMS provides a means of doing just that: *Mount Verification*.

Mount verification is the mechanism whereby units are brought back online, and the volumes in those units validated, after a serious (but hopefully recoverable) problem has rendered the volumes inaccessible to a host VAX.

In a very general sense, performing mount verification for a disk drive consists of two major tasks:

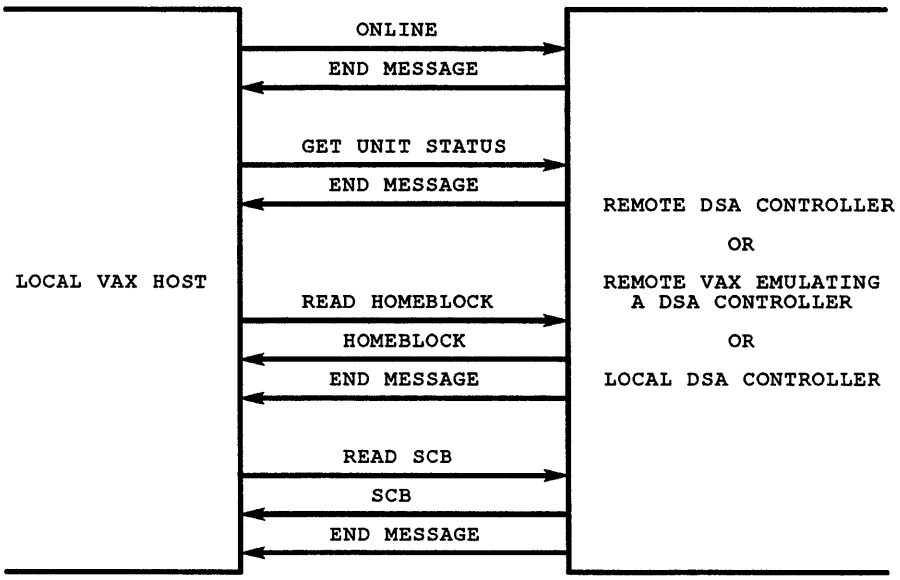
- The first of these tasks is bringing the unit back online to the local host.

  This is done by the mount verification routines in module *MOUNTVER* issuing an *IO$_PACKACK* (pack acknowledge) function to DUDRIVER.

  The disk class driver implements this function as a three-step operation:
  - — DUDRIVER locates a path to the disk through "some" controller.
  - — It then issues an MSCP ONLINE command to the controller to bring the unit online to the host (and to the controller if it isn't already).
  - — Finally, if the ONLINE is successful, it issues a GET UNIT STATUS to obtain the status and geometry of the unit.
- The second major task is verifying that the volume in that unit is the same volume that was there before the occurrence of the event requiring this recovery procedure.

  This is a two-step operation. First, the Homeblock is read, and selected fields are compared against previously stored values: checksum, volume serial number, and volume name. Then, the Storage Control Block (SCB) is read, and some of its fields are also compared against previously stored values: checksum, mount time, and volume lock name. For shadow set units, these checks are made from the information in the virtual VCB. This general flow is depicted in Figure 4–6.

Figure 4–6: Mount Verification Validation Flow



NOTE

There are no MSCP commands specifically for reading the Homeblock or SCB. Shown in the diagram above are MSCP read commands set up to specifically reference those disk blocks.

Mount verification is not that simple. A number of very complex issues must be addressed as part of the total procedure:

- To which disks can mount verification be applied?
- What circumstances lead to mount verification, and from what conditions is it possible for mount verification to recover?
- What is done with new I/O requests for disks while they undergo mount verification?
- What about I/O requests which have been partially processed by DUDRIVER?
- What about requests for which MSCP commands have been issued, but for which end message have not yet been received?

Then, of course, there is the topic of failover for dual-pathed disks. This in itself presents still more complications. How does DUDRIVER find an alternate path to the disk? How does it actually trigger failover of the disk once it has found an alternate (i.e. secondary) path?

This section delves into the topic of mount verification as it pertains to disks on "MSCP speaking" controllers. For now, the discussion is confined to non-shadowed disks. Volume shadowing is covered in a later chapter.

4.5.1 Circumstances Leading to Mount Verification

Circumstances leading to mount verification can be grouped into a few basic categories.

- The I/O request completion routine, *IOC$ALTREQCOM*, calls *EXE$MOUNTVER* to perform mount verification for a particular disk when a read or write request for that disk completes unsuccessfully. This is characterized by the controller for the disk returning an MSCP end message with a major status code indicating an error.

  Upon receipt of an end message, DUDRIVER extracts the RSPID from the message buffer. Using the low order 16 bits of the RSPID as an index into the *Response Descriptor Table* (RDT), it locates the RDT entry containing the address of the CDRP. The CDRP contains the preserved context of the I/O request with which the end message is associated.

  It then resumes the request at the PC saved in the CDRP. The resumed request immediately tests the major status code; seeing that the status code indicates an error, the request branches to TRANSFER_MSCP_ERROR where it converts the MSCP status code into an equivalent VMS condition value.

  After doing appropriate error logging and releasing of SCS resources, a branch is made to IOC$ALTREQCOM. IOC$ALTREQCOM sees that the VMS condition value reflects an error; so it passes the associated IRP and UCB to EXE$MOUNTVER. There mount verification begins for the particular unit associated with the UCB.

  One exception to this scenario would be if the FLAGS field indicates that the host needs to perform bad block replacement on a disk handled by a local DSA controller. Then a branch is taken from DUDRIVER to *XFER_REPLACE* instead of invoking mount verification. This exception would not happen with an HSC or a remote VAX emulating an "MSCP speaking" controller; they handle bad block replacement themselves without local host intervention.

  The second exception occurs when the major status code field contains an "unexpected" quantity, something which DUDRIVER is not prepared to handle. This leads to the connection being broken with the server in routine *INVALID_STS* and leads into the next category of events causing mount verification.

- The SCS connection between the local disk class driver and the controller's MSCP disk server is broken for any reason. There are many events which may lead to this.

  All "unexpected" major status codes in an MSCP end message are considered "invalid". DUDRIVER considers the controller to be "very ill" for issuing such a status code. Therefore, end message processing branches to routine *DU$RE_SYNCH* to reset the controller. There are two possible scenarios:

  — If the controller is either an HSC or a local DSA controller, then DU$RE_SYNCH does a "host clear" of the controller by sending it an MSCP RESET followed by an MSCP START. This causes the controller to reinitialize, thus breaking the SCS connection between the local DUDRIVER and the controller's MSCP disk server.

When the local host's SCS layer within the local port driver discovers the loss of the connection, it invokes DUDRIVER's connection error handling routine, DU$CONNECT_ERR. One of the tasks performed by this routine is to call *DUTU$REVALIDATE*, which, in turn, calls *EXE$MOUNTVER* for each "qualified" disk unit for which the controller was providing the primary path.

DUTU$REVALIDATE is passed the CDDB associated with the controller in question; and it passes to EXE$MOUNTVER a UCB from the list of UCBs attached to the CDDB each time it calls EXE$MOUNTVER.

Each time EXE$MOUNTVER is called in this scenario, it is passed the UCB associated with the unit for which mount verification is being requested. Since this scenario is for a "sick" controller, disks are being submitted to mount verification in potentially large numbers. A "dummy" IRP is provided to EXE$MOUNTVER with a status of SS$_DEVOFFLINE each time it is called to facilitate coordination with the class drivers mount verification routine DUTU$MOUNTVER.

— If the controller is actually a remote VAX running the VMS based MSCP server, then DU$RE_SYNCH does not attempt a "host clear". It merely branches into common code within DU$CONNECT_ERR, intentionally DISCONNECTs from the remote VAX's MSCP server, and then calls DUTU$REVALIDATE. From this point on, this scenario is essentially the same as the first.

Remember that other events can also lead to the calling of DU$RE_SYNCH, and thus also to mount verification. Some examples would be no progress in the oldest command still outstanding for a controller, timeout of immediate class commands sent to a controller, and receiving an invalid attention message from a controller.

The calling of DU$RE_SYNCH intentionally breaks the SCS connection with the server. However, the connection breaking could be a consequence of numerous unexpected failures as well. Perhaps a controller had an unrecoverable failure and crashed by itself. Prolonged noise on the CI causing a protocol failure between ports could cause the ports at both ends to close the virtual circuit supporting the connection, and hence break the connection.

Similarly, the failure of the port hardware at either end of a virtual circuit would cause the port driver to close the circuit, and again the connection would be broken. All such cases would lead to DU$CONNECT_ERROR, and from there to DUTU$REVALIDATE.

- A third category might be labeled "connection manager induced" mount verification.

  If the local host is a member of a VAXcluster, then the connection manager uses mount verification to stall disk activity if either quorum has been lost, or if a quorum file read or write operation completed with a "media offline" or "volume invalid" error.

  The connection manager accomplishes this by calling DUTU$REVALIDATE from within routine *EXE$CLUTRANIO* for each "MSCP speaking" controller that the local host can see. Each time DUTU$REVALIDATE is called, it is passed a CDDB from the list of CDDBs managed by DUDRIVER. The head of this list is at *IOC$GL_DU_CDDB*.

4.5.2 Disks Which Qualify for Mount Verification

DUTU$REVALIDATE and EXE$MOUNTVER apply different criteria for qualifying a unit on an "MSCP speaking" controller for mount verification. When EXE$MOUNTVER is called directly, its criteria are the sole determining factors in making this determination. But when called by DUTU$REVALIDATE, then EXE$MOUNTVER may be thought of as refining the criteria already applied by DUTU$REVALIDATE.

If the local VAX is not in a cluster, or if quorum has not been lost, then the unit must satisfy all of the following criteria to be considered qualified by DUTU$REVALIDATE:

- It was previously software valid (i.e. its homeblock and storage control block previously validated correctly).
- The unit is not a controller based shadow set member. (This does not preclude a shadow set virtual unit. Shadow set members are handled within the context of the virtual unit to which they belong, and not by themselves.)

  If the unit is a Host based shadow set member, special processing will be performed by routine *REVAL_HBS_MEMBER*
- The unit is not already in mount verification.
- For *Path Move* requests, minimal mount verification processing will be performed as with the quorum lost processing.

Within the context of a VAXcluster, more than one VAX may be accessing a unit if it is either dual-pathed or attached to a multihost controller. It is therefore necessary for the local host to stall its I/O requests to such a unit if it loses quorum. In the event that quorum is lost, DUTU$REVALIDATE also considers a unit qualified if, in addition to all the above criteria, it is either dual-pathed or on a multihost controller.

NOTE

The multihost criteria qualifies disks which are MSCP-served by remote hosts.

In general, the criteria applied by EXE$MOUNTVER are as follows:

- There must be a volume mounted in the unit.
- The volume must not be mounted "foreign".
- Mount verification must be enabled for the unit. (This is the default when a disk is mounted as a file structured volume.)
- The error which led to EXE$MOUNTVER being called must be one from which mount verification can recover. These include the following:

Error Code	Description
SS$_MEDOFL	Media offline
SS$_VOLINV	Volume invalid
SS$_DEVOFFLINE	Device offline (because controller is inoperative)
SS$_SHACHASTA	Shadow set state/membership change
SS$_WRITLCK	Volume is writelocked

However, if EXE$MOUNTVER is triggered by the connection manager via EXE$CLUTRANIO, it does not bother to apply these criteria; it goes directly to mount verification.

There is a special case when disks mounted foreign are subjected to a "limited degree" of mount verification. This occurs when the connection manager requests mount verification from within routine EXE$CLUTRANIO. The sole purpose of this is to block I/O to such disks until quorum is regained. Even though the IO$_PACKACK function is performed under these circumstances, the Homeblock and Storage Control Block are not validated.

4.5.3 Failover of Dual-Pathed Disks

Consider a situation wherein a disk, DJA1, is dual-pathed between two HSCs. This *dual-pathing* is "static"; all the I/O from any host in the cluster is directed through the HSC which is currently serving as the primary path controller for the disk. The other HSC is there to handle the disk only if the primary HSC is unable to.

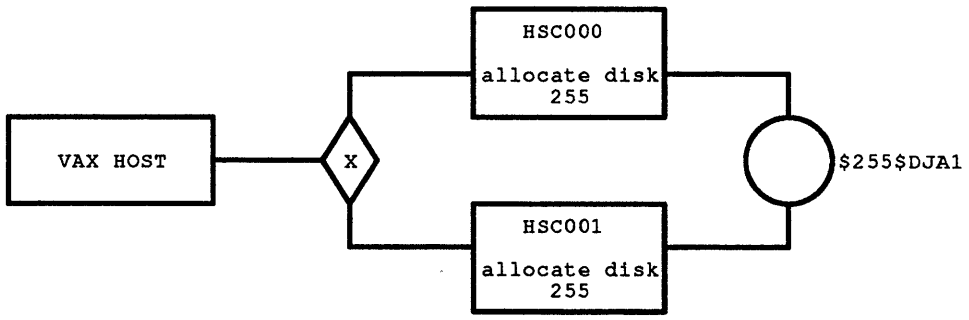
In this situation, a process need not be concerned with which HSC is currently handling the disk. All that needs to be done is to assign both HSC's to the same *Allocation Class*, say 255. Then the process merely refers to the disk as 255DJA1.

By using the allocation class format of a device name, the process is instructing the operating system to route I/O requests through the proper controller; but how that's done is transparent to the process. Figure 4–7 illustrates a DSA device which is dual pathed between two HSCs.

NOTE

It is often considered desirable to make even the allocation class transparent to the user process through the use of a logical name.

Figure 4–7: Dual Pathed Disk Device Configuration



Based on the previous chapters covering data structures and the flow of a $QIO, the following rules of setting the allocation class must be observed:

- The two nodes providing the dual-pathing for a disk must be in the same nonzero allocation class.

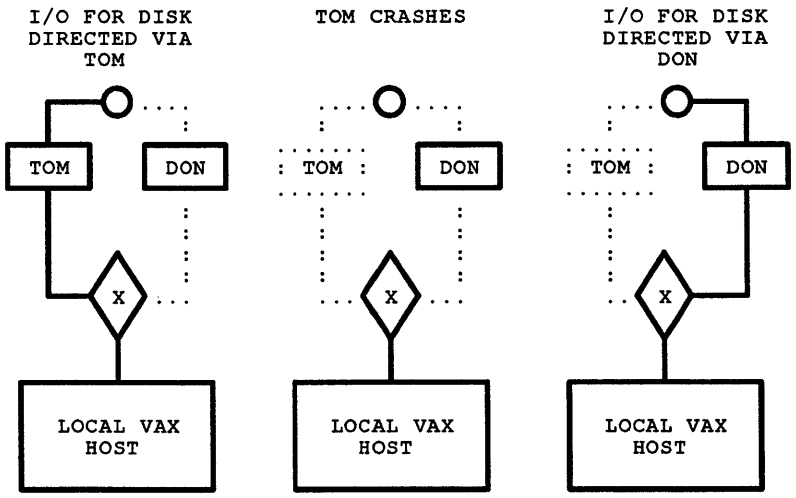
  If they weren't, the disk would appear as two disks. For example, if one HSC were assigned allocation class 255, and the other 254, then there would appear to be two disks: one called 255DJA1 and another called 254DJA1. The range of valid allocation classes as of this writing is 1 through 255. The number zero is used to indicate that the controller is not assigned an allocation class.
- All cluster accessible disks on nodes with the same nonzero allocation class must have unique names.

  Suppose that two VAXes are both in allocation class 1 and run the VMS based MSCP server. It is invalid for each VAX to have, and to set served, its own MASSBUS disk DRA3. If this were done, then it would be ambiguous as to which of the two disks is being referenced by the name 1DRA3.
- Restricted access disks that are not cluster accessible can have the same name on nodes which are in the same allocation class. The key point is that these disks *MUST NOT* be dual-ported and *MUST NOT* be set served.
- For a CI VMS disk server to serve HSC based disks to NI cluster members, the allocation class of the VMS node must match that of the HSC.

Assume that an event occurs rendering a dual-pathed disk inaccessible through its current primary path controller. Then mount verification will attempt to fail it over to the alternate path controller. What was the alternate path controller now provides the primary path; and what was the primary path controller becomes secondary.

As an example, consider an RA60 dual-pathed between two HSC90s called TOM and DON, with TOM being the primary path controller. If TOM crashes, then mount verification will failover the disk to DON. DON is now the primary path controller, and TOM is secondary. All I/O to the RA60 is directed through DON. Figure 4–8 depicts the stages of Failover for a dual pathed HSC disk.

Figure 4–8: Failover of a Dual Pathed HSC disk



If sometime after TOM reboots, power is cut to DON, then mount verification will failover the disk back to TOM, making TOM the primary controller again.

The event causing failover does not have to be as catastrophic as a controller completely failing. Perhaps one *Standard Disk Interface* (SDI) module in an HSC fails, but the HSC survives; then only the disk units on that SDI would failover. It could even be caused by someone "popping out" the primary path port button on the disk itself.

Consider a "generic event" that inhibits communication between controller TOM and the RA60 disk. After a very short (but disk and event dependent) time period, the disk gives up trying to re-establish communication with TOM, and sets itself "available" to DON. Meanwhile, the local host VAX becomes "aware" of the problem. Perhaps TOM sent an end message with a major status of OFFLINE and a sub-status of INOPERATIVE. Or perhaps TOM crashed, and the connection between the local disk class driver and TOM's MSCP disk server was broken.

Regardless of which of these events made the local VAX aware of the problem, both of them will lead to mount verification for the disk. The key difference to note between them is that with the first, there is still an SCS connection between DUDRIVER and TOM's MSCP server.

Assuming the connection with TOM's server is still active, mount verification will direct the first ONLINE to controller TOM. But since TOM can't communicate with the unit, TOM returns an end message with a major status code of OFFLINE and sub-status code of INOPERATIVE. This will then cause mount verification to arrive at the same point it would have had the connection with TOM's server been broken. An alternative controller must be found which can both communicate with the disk and is in the same allocation class as TOM.

If the local host already knows about such an alternative controller, then secondary path linkages will already have been set up in the unit's UCB for this controller. If not, then all CDDBs corresponding to controllers other than TOM are scanned for those in the same allocation class as TOM. As each such controller is found, to it is sent a GET UNIT STATUS command containing the unit number of the RA60. The first controller to respond with an end message bearing the major status code of AVAILABLE or ONLINE is considered as providing a viable secondary path for the disk.

Assume that controller DON is selected, either because it provides an already known secondary path, or because it was the first to respond properly during the CDDB scan. The primary and secondary path linkages in the UCB are adjusted to reflect that DON is now providing the primary path, and that TOM is now providing the secondary path.

Then the ONLINE is issued to DON to actually bring the RA60 online to the local host (and to controller DON if it isn't already). If the ONLINE succeeds, a GET UNIT STATUS is issued to DON to verify that the RA60 is still ONLINE, and to obtain the disk's status and geometry.

4.5.4 Mount Verification Volume Validation

At the very core of the mount verification procedure is a routine, *PERFORM_VALIDATE*, invoked from EXE$MOUNTVER. By calling two subroutines, PERFORM_VALIDATE executes the two major tasks that represent the very heart of the entire process:

- *PACKACK_VOLUME*

  This routine issues the IO$_PACKACK function (ONLINE and GET UNIT STATUS) to establish a path to the disk, and to bring it online to the local host. Failover, if it occurs, happens internal to the IO$_PACKACK, and is transparent to PERFORM_VALIDATE.

  PERFORM_VALIDATE doesn't really care whether or not failover occurred; all it cares about is that a path has been restored to the disk. In fact, if PACKACK_VOLUME fails in its task, it doesn't return to its caller (PERFORM_VALIDATE); instead it aborts mount verification for the unit in question.

- *VALIDATE_VOLUME*

  The second major task is verifying that the volume currently present in the unit is the same volume present before mount verification became necessary. This is done by routine VALIDATE_VOLUME reading the Homeblock and Storage Control Block (SCB), and then comparing their contents against previously stored information.

  The items of comparison are as follows:

  — Checksums for both the Homeblock and SCB.

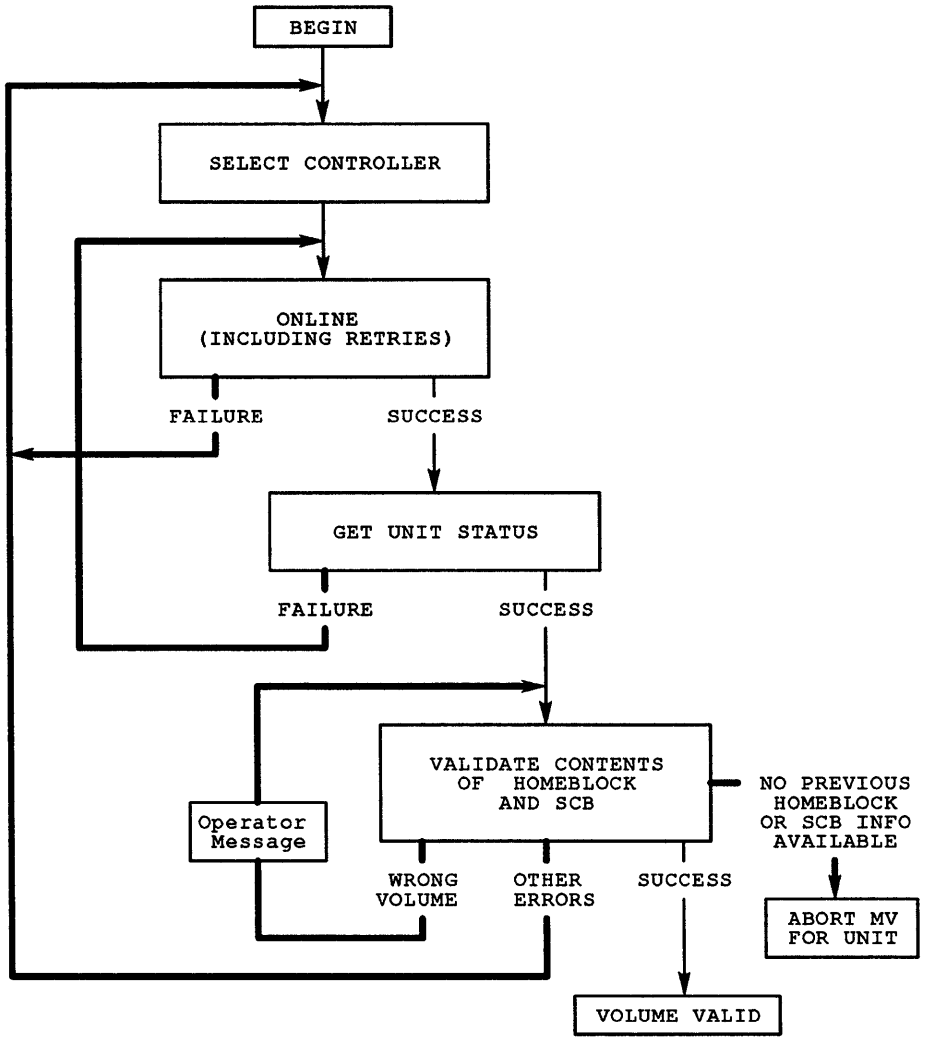
  — Volume serial number and name (Home Block)

  — Mount time (Storage Control Block)

  — Volume lock name (If Volume Name did not compare)

What follows is an overview of the inner workings of these routines. It is supported by Figure 4–9. This illustration is intended as a simplified block diagram. As such, it integrates the activities of the three routines into one logical flow, and emphasizes the chronology of events. It is intended to assist in following the overview which, on the other hand, is concerned with the specifics of the routines.

Figure 4–9: Volume Validation Flow



4.5.4.1 Perform_Validate Routine

PERFORM_VALIDATE begins by sending an *OFFLINE* message to the OPCOM process as well as explicitly to device OPA0:, notifying any operator terminals of a device offline condition. It then calls routine PACKACK_VOLUME to bring the unit back online to the local host.

4.5.4.2 PACKACK_VOLUME Routine

PACKACK_VOLUME issues an IO$_PACKACK request to DUDRIVER's start I/O routine. Servicing this request begins with the selection of a controller to which to send an ONLINE command. The rules for selection work as follows:

- If the SCS connection with the current primary path controller has not been broken, then that controller is selected.
- If the SCS connection with the current primary path controller had been broken, then an alternate controller is selected by calling routine *DUTU$LOCATE_UNIT* which will in turn call routine *DUTU$BEGIN_CONN_WALK*. This routine will determine if an existing secondary path is valid and if not will initiate a *Connection Walk* to locate a valid path.

  The steps performed by the Connection Walk routine are as follows:

  — Check to see if a *Preferred Path* has been specified by examining the *UCB$L_PREF_CDDB* field of the UCB.

  If a preferred path has been requested, clear the *Load Balance* flag in the CDRP and attempt to validate the path. If valid, the path will be used. If the preferred path is not valid, a connection walk will be started to find a valid path.

  — Check to see if a *Load Balance* request has been made. If so, branch to perform connection walk.

  — Check the UCB to see if secondary path linkages have already been set up. If so, then select the controller associated with that secondary path.

  — If a secondary path is not yet set up, then scan the list of CDDBs looking for controllers in the same nonzero allocation class as the current primary path controller. If one is found, determine if it can access the unit by sending it a GET UNIT STATUS command. If the controller replies that the unit is either already ONLINE to the controller, or at least AVAILABLE to it, then that controller is selected. If not, continue the scan until a controller is selected or the CDDB list is exhausted.

  If an alternate controller is selected by either of the these two mechanisms, then the UCB path linkages are modified such that the current primary path controller becomes secondary, and the alternate controller becomes primary.

If the IO$_PACKACK fails to select a controller, then PACKACK_VOLUME merely tries it again after a pause of approximately one second. In fact, PACKACK_VOLUME loops here in the hope that either an SCS connection with the current primary path controller is re-established, or the disk enters the ONLINE or AVAILABLE state with some other controller with which the local host has an open SCS connection.

NOTE

This loop is not infinite. It does have a timeout mechanism of waiting *MVTIMEOUT* seconds. This is presented in Section 4.5.5.

If the IO$_PACKACK does select a controller, then it issues an ONLINE for the unit to the controller. This operation has three possible outcomes:

- The end message indicates a failure with the major status code OFFLINE and sub-code of either UNKNOWN or INOPERATIVE.

If this is the case, then the alternate controller selection mechanism described above is employed to select a different controller. If one is found, then the UCB primary and secondary path linkages are modified (also as described above), and the ONLINE is retried with the new primary path controller.

NOTE

This approach is specifically designed to handle the situation where the SCS connection with the original primary controller survived, but the controller itself can't communicate with the disk. It attempts to force failover to some other controller, even though the original primary controller is still "alive".

Now, both primary and secondary path linkages are established. So it can be seen that ONLINE attempts can *ping pong* between two controllers so long as the same major status and sub-status codes remain OFFLINE and either UNKNOWN or INOPERATIVE, respectively. Thus, a retry limit of 4 is built into this loop for VMS V5.5. This limit is enforced in routine *PACKACK_UNKNO_INOPR* in module dudriver.lis.

If the retry limit is exhausted, the IO$_PACKACK returns to PACKACK_VOLUME with an error. This merely causes PACKACK_VOLUME to retry the IO$_PACKACK again after pausing approximately one second.

- The IO$_PACKACK completes with any other error. Again, PACKACK_VOLUME will merely retry the IO$_PACKACK after pausing approximately one second.

NOTE

Observe that PACKACK_VOLUME will switch between controllers as the UCB's primary and secondary linkages are interchanged with retries of the IO$_ PACKACK operation.

- The ONLINE is successful. Then the IO$_PACKACK continues by sending a GET UNIT STATUS command to the controller.

As already stated, the purpose of the GET UNIT STATUS command is to obtain both geometry and status information about the unit. The outcome here is binary: it either fails or succeeds. If it fails, the unit is assumed to have gone OFFLINE relative to the controller, and IO$_ PACKACK merely branches back to to the point where it issues an ONLINE. If it succeeds, then the IO$_PACKACK completes, returning a "success" status code to PACKACK_VOLUME. PACKACK_VOLUME then returns to its caller, PERFORM_VALIDATE.

Next, PERFORM_VALIDATE executes its second major task by calling routine VALIDATE_ VOLUME. It wants to make sure that the volume now in the unit is the same one that was there before.

4.5.4.3 VALIDATE_VOLUME Routine

VALIDATE_VOLUME first reads the Homeblock and validates the Homeblock's checksum, the volume serial number, and the volume name. It then reads the SCB and validates the SCB's checksum, mount time, and volume lock name (if appropriate). There are four possible outcomes here:

- If both the Homeblock and SCB validate successfully, then VALIDATE_VOLUME merely returns to its caller, PERFORM_VALIDATE. The volume and unit are considered VALID.
- If wrong volume information is found in either the Homeblock or the SCB, then VALIDATE_VOLUME pauses approximately one second, and loops back to try again. The operator is also notified of the problem.
- If previous information is not available against which to check the contents of the Homeblock and SCB, then VALIDATE_VOLUME does not return to its caller. Instead, it aborts mount verification for the unit, leaving it *Software Invalid.*
- VALIDATE_VOLUME returns all other errors to its caller, PERFORM_VALIDATE. This causes PERFORM_VALIDATE to loop back to its beginning and start over by again calling PACKACK_VOLUME.

4.5.5 Mount Verification Timeout

Once started, mount verification for a unit could turn into an infinite loop if some mechanism were not explicitly provided to prevent it. The Sysgen Parameter MVTIMEOUT (mount verification timeout period) provides that mechanism.

EXE$MOUNTVER allocates an internal IRP for use by PERFORM_VALIDATE when requesting the IO$_PACKACK, and for reading the Homeblock and SCB. Just before entering PERFORM_VALIDATE, EXE$MOUNTVER computes the sum of the current time of day and the value of the MVTIMEOUT parameter, and stores the sum in the IRP. This sum is called the Mount Verification Timeout.

Each time PACKACK_VOLUME pauses before retrying the IO$_PACKACK, it checks to see if the current time of day has progressed past the mount verification timeout for the IRP. If so, it aborts mount verification for the unit.

The value of MVTIMEOUT is stored in location *IOC$GW_MVTIMEOUT*. As of VMS V5.5, it defaults to *3600 seconds* (i.e. 60 minutes). It is desirable that this parameter be large enough to insure controllers have enough time to reinitialize, and also re-establish SCS connections, before mount verification times out any disks they may have which are not dual-pathed. 60 minutes is currently considered reasonable for this.

4.5.6 Disks Requiring Special Handling

Certain disks require special handling, namely system disks, quorum disks, and disks which are mounted foreign.

4.5.6.1 Foreign Disks

Unless invoked by the connection manager, EXE$MOUNTVER does not perform mount verification on foreign disks. At its very beginning, it detects the DEV$V_FOR flag set in the UCB's DEVCHAR field and immediately returns to its caller.

If invoked by the connection manager, then mount verification of foreign disks is permitted to progress up through routine PACKACK_VOLUME. Routine PERFORM_VALIDATE notices that the disk is foreign, and thus knows that there is no information with which to compare the contents of the Homeblock and the SCB. PERFORM_VALIDATE will fake successful validation as it bypasses the call to VALIDATE_VOLUME and branches directly to the routine to wait for quorum to be regained. Again, remember that the reason for doing this is to block I/O to the foreign disk until the host is sure that it has quorum.

4.5.6.2 System Disk and Quorum Disk

PERFORM_VALIDATE must treat the system disk and the quorum disk the same way if they have not yet been properly mounted. It doesn't have any previous information to compare the Homeblock and SCB with, so again it bypasses the call to VALIDATE_VOLUME.

4.5.7 Stalling and Unstalling I/O During Mount Verification

When one looks at the circumstances which induce mount verification for a unit, one can say that they all place the data on the volume "at risk". It is therefore necessary to stall I/O requests for the unit until the volume is no longer at risk.

For purposes of this discussion, I/O requests can be classified in terms of how far they have progressed since they were issued. In this sense, there are two groups of I/O requests:

- There are those requests whose IRPs were not passed to DUDRIVER until after mount verification for the unit began. These are the easiest to deal with and are called *New I/O requests.*
- There are those requests whose IRPs were passed to DUDRIVER before mount verification began. They are at various stages of processing, but have not yet been completed; so they are called *Active I/O requests.* Handling these is more complicated.

The key to dealing with new I/O requests is a field in the UCB called the *Resource Wait Count* (RWAITCNT) field. Before entering PERFORM_VALIDATE, EXE$MOUNTVER calls a driver specific "begin mount verification" routine. For DUDRIVER, this is *DUTU$BEGIN_MNTVER.* Two of the tasks it performs are:

- Set the *UCB$V_MSCP_MNTVERIP* flag in the UCB, indicating that MSCP specific steps for mount verification have begun.
- Increment the RWAITCNT field in the UCB.

Now when new I/O requests are handed to DUDRIVER, its start I/O routine sees the RWAITCNT field is nonzero. So the IRP is inserted into a queue on the UCB (*UCB$L_IOQFL*), and there it stays until mount verification completes. This queue is called the *Pending IRP* or *Pending I/O Queue.*

Regardless of whether or not the volume is declared "Software Valid", mount verification for disks on "MSCP speaking" controllers passes through DUDRIVER's End Mount Verification Routine, *DUTU$END_MNTVER*.

Within that routine, the UCB's RWAITCNT field is decremented. This permits the unstalling of pending IRPs. Of course, if mount verification was unable to restore the volume to the VALID state, then all such IRPs are terminated.

NOTE

The actual unstalling of requests in the pending IRP queue is done by *SCS$UNSTALLUCB*, which is called whenever the RWAITCNT field is decremented. SCS$UNSTALLUCB tests the RWAITCNT field. If the field contains a zero, then this routine removes the IRPs from the queue and passes them to the driver's start I/O routine. If the field is still nonzero, then I/O is still stalled on this unit; so SCS$UNSTALLUCB merely returns to its caller without doing anything. One of those reasons will be presented momentarily.

Various special processing steps must be taken when handling active I/O requests:

- One of those steps involves the MSCP_MNTVERIP flag set by DUTU$BEGIN_MNTVER in the UCB.

  End messages received for active I/O requests and bearing major status codes indicating errors are passed along with their associated IRPs to EXE$MOUNTVER, as already described. When DUTU$BEGIN_MNTVER gets called for these, it sees that mount verification is already in progress; the MSCP_MNTVERIP flag is already set. So it merely inserts these IRPs into the pending IRP queue along with the new IRPs.

  So as to maintain integrity of the order of I/O operations, active IRPs are inserted into this queue in *Sequence Number Order*.

  NOTE

  New IRPs are merely inserted at the end of the queue as they are received by DUDRIVER. This preserves the natural order among the new IRPs, and also relative to any active IRPs in the queue.

- If mount verification is triggered by the failure of the connection between DUDRIVER and the controller's MSCP disk server, then disk failover presents its own special problem.

  The disk class driver's connection error routine, DU$CONNECT_ERR, gathers up all active CDRPs (and hence active IRPs) from wherever they are, and inserts them in the controller's CDRP restart queue. This is done before DU$CONNECT_ERR calls DUTU$REVALIDATE, which in turn calls EXE$MOUNTVER for each unit on the controller.

  NOTE

  There is a CDRP sequence number field used to preserve the proper order of the requests.

If a disk is failed over to an alternate controller, its active CDRPs (which are really active IRPs) are removed from the old controller's CDRP restart queue and inserted at the head of the UCB's pending IRP queue.

- There is another problem created for mount verification when it results from the loss of an SCS connection with a controller's disk server. If a connection is re-established, then the active requests in the controller's CDRP restart queue must all be processed prior to unstalling any UCB's pending IRP queue.

  Here again, the UCB's RWAITCNT field is the key. In the common code they share, DU$RE_SYNCH and DU$CONNECT_ERR increment the RWAITCNT field of each UCB on the controller if it has not yet been incremented for mount verification. Thus, when DUTU$BEGIN_MNTVER increments this field, it will contain at least a 2. Later, DUTU$END_MNTVER will decrement it; and then it will contain at least a 1. When SCS$UNSTALLUCB is called, it will see the RWAITCNT field is nonzero; so it will return to its caller without unstalling the UCB's pending IRPs.

  As mount verification completes for each unit on a controller, the count of the number of UCB's waiting for completion is decremented. When this count goes to zero, *DUTU$RESTART_NEXT_CDRP* begins single stream processing of the CDRPs remaining in the controller's CDRP restart queue. When single stream processing completes for a controller, the RWAITCNT field of each of its UCBs is decremented and SCS$UNSTALLUCB is called for that unit. This time, SCS$UNSTALLUCB should find the RWAITCNT field zero; so now it can unstall the requests in the UCB's pending IRP queue.

4.5.8 Aborting Mount Verification

If for any reason mount verification for a unit must be aborted, such as mount verification timeout, then the following actions are taken:

- The operator is notified that mount verification for the unit has been aborted.
- The VALID flag in the unit's UCB is cleared.
- Mount verification is disabled for the unit.
- I/O requests for this unit in the controller's CDRP restart queue are terminated with the VMS condition value *SS$_VOLINV* (Volume Invalid).
- I/O requests in this unit's pending IRP queue (UCB$L_IOQFL) are terminated with the VMS condition value SS$_VOLINV.

4.5.9 Mount Verification - The Big Picture

To tie together the various pieces of mount verification, here is a general outline of the steps taken by routine EXE$MOUNTVER when performing mount verification for a disk on an "MSCP speaking controller". EXE$MOUNTVER will be passed a "dummy" IRP if it is called from DU$RE_SYNCH or DU$CONNECT_ERR. If called for a single unit due to an error in a "real" I/O request, then the IRP for that request is passed to EXE$MOUNTVER. In both cases, it is also passed the UCB of the unit for which mount verification is being requested.

Remember that if mount verification is invoked by DU$_RESYNCH or DU$CONNECT_ERR, then the RWAITCNT field of the UCB will have been incremented once before entry into EXE$MOUNTVER.

- Determine if mount verification possible and necessary for unit.
  - — NO - Return to caller. (SS$_xxx condition value in IRP unaltered.)
  - — YES - Continue with next step.
- If mount verification not already in progress, allocate special MVIRP (mount verification IRP) to be used later for IO$_PACKACK, reading Homeblock, and reading SCB (Storage Control Block).
- Perform MSCP specific "begin mount verification tasks"
  - — If MSCP_MNTVERIP flag in UCB is zero, then
    - o Set MSCP_MNTVERIP flag.
    - o Increment RWAITCNT field in UCB.

NOTE

New I/O requests are now stalled. Their IRPs will be inserted onto the UCB's pending IRP queue when passed to DUDRIVER until mount verification completes.

- If IRP passed to EXE$MOUNTVER represents a "real I/O" request (and is not an "internal" IRP, then insert it onto the UCB's pending IRP queue.
- If an MVIRP was not allocated above, then
  - — Abort mount verification if failure to allocate due to lack of nonpaged pool. (SS$_VOLINV)
  - — Return to "caller's caller" if MVIRP not allocated because mount verification was already in progress for this unit.

NOTE

The only effect of calling EXE$MOUNTVER if mount verification is already in progress will be the insertion of an IRP representing a "real I/O" request onto the UCB's pending IRP queue.

If an MVIRP was allocated above, then

- — Calculate and store in IRP the mount verification timeout.
- Perform volume validation by
  - — Issuing an IO$_PACKACK function
    - o Sends ONLINE and GET UNIT STATUS commands

NOTE

Brings unit online to host (and controller if not already).

- o Failover unit to alternate controller if necessary and possible.

— Read and verify contents of Homeblock and SCB.

NOTE

Make sure that volume currently in unit is same volume that was there before mount verification was necessary.

There are four possible outcomes of the this step:

— If no previous volume information available, then abort mount verification for this unit (SS$_VOLINV).

— If wrong volume found in unit, issue message and loop until operator resolves problem (SS$_INCVOLLABEL).

— For any other errors, repeatedly retry this step until it succeeds, or until mount verification times out for this unit. If mount verification times out, then abort mount verification for the unit (SS$_VOLINV).

— If success, then continue with next step.

NOTE

The unit and volume are now considered "Software Valid".

- Wait for quorum to be regained if it was lost.
- Perform MSCP specific "end mount verification" tasks.
  — Clear MSCP_MNTVERIP flag.
  — Decrement UCB's RWAITCNT field.
  — If unit not software valid, then terminate I/O requests (SS$_VOLINV) for unit in controller's CDRP restart queue and UCB's pending IRP queue.

  If unit is software valid, then unstall new I/O requests in UCB's pending IRP queue if RWAITCNT field is zero.
- Decrement CDDB's WTUCBCTR field (number of UCBs on controller waiting for mount verification to complete).

  If WTUCBCTR field is now zero and CDDB's CDRP restart queue is not empty, then restart CDRPs in CDDB's CDRP restart queue.

NOTE

After handling last CDRP in restart queue, RWAITCNT field will be decremented to zero and I/O requests in pending IRP queue unstalled.

4.5.10 Mount Verification Routines

The next few sections present detailed descriptions of the major routines involved with mount verification of non-shadowed disks on "MSCP speaking" controllers (including VAXes running the VMS based MSCP server).

Routine Name	Routine Description
DUTU$REVALIDATE	DUDRIVER's top level routine called to perform mount verification for all "qualified" disks on a controller.
EXE$MOUNTVER	Handles mount verification for a single disk unit.
PACKACK_VOLUME	Initiates the IO$_PACKACK function required during mount verification of a disk unit.
EXE$MNTVERSIO	Passes mount verification IRPs to DUDRIVER.
DUTU$RESTART_NEXT_CDRP	Restarts I/O requests in a CDDB's CDRP restart queue in single stream mode.
DUTU$END_SINGLE_STREAM	Ends single stream mode processing of CDRPs after CDDB's CDRP restart queue is empty, and calls SCS routine to unstall I/O requests in UCB's pending IRP queue.

4.5.10.1 DUTU$REVALIDATE

Routine DUTU$REVALIDATE is the top level routine called to re-validate all disks on an "MSCP speaking" controller. In these flows, the term controller will apply to any local DSA controller, an HSC, an ISE, or a remote VAX which is running the VMS based MSCP server.

The times when this routine are called fall into two general categories:

- The SCS connection between the local disk class driver and the MSCP server on the controller has failed. Then DUTU$REVALIDATE is called from either DU$RE_SYNCH or DU$CONNECT_ERR.
- The connection manager calls this routine from within EXE$CLUTRANIO to stall disk activity if the local host is a member of VAXcluster and either
  - — it has lost quorum, or
  - — a quorum file read or write operation completed with a "media offline" or "volume invalid" error.

  This category is distinguished from the first by EXE$CLUTRANIO setting to 1 the QUORLOST flag in the CDDB$W_STATUS field before calling DUTU$REVALIDATE.

The basic input for this routine is the CDDB corresponding to a controller. Thus, for the first category, DUTU$REVALIDATE is called only for a particular controller. However, for the second category, EXE$CLUTRANIO calls it for each controller visible to the local VAX. Regardless of the situation, DUTU$REVALIDATE will perform mount verification for all disks units being served by each controller for which it is called.

The list of UCBs attached to the CDDB is scanned. The listhead is at offset CDDB$L_UCBCHAIN, and the linkage from one UCB to another is provided by each UCB's CDDB_LINK field.

- If a disk is not qualified for mount verification, this routine skips it and goes on to the next.

  If the local VAX is not in a cluster, or if quorum has not been lost, then a unit qualifies for mount verification if the following conditions all apply to it:

  — The VALID flag must be set in its STS field. This indicates that the volume was previously software valid.

  — The SSM flag must not be set in its DEVCHAR2 field. The unit must not be a Member of a *Controller Based Shadow Set*.

  NOTE

  The UCB representing the entire shadow set virtual unit does not have its SSM flag set; only the member UCBs do. So a shadow set virtual unit UCB passes this test.

  — If this is a member of a *Host Based Shadow Set* branch to special routine *REVAL_HBS_MEMBER*.

  — The MSCP_MNTVERIP flag must not be set in its DEVSTS field. (This flag being set indicates that the disk has already entered the MSCP specific stages of mount verification.)

  — If this is a Pathmove request, signal minimal mount verification required by setting the UCB$V_CLUTRAN bit in the UCB$L_STS field.

  If the local VAX is in a cluster, then some other VAX may be accessing a unit the local VAX is using if either

  — The unit is dual-pathed (the 2P flag is set in the UCB$L_DEVCHAR2 field).

  — The unit is on a multihost controller such as an HSC (the CF_MLTHS flag is set in the CDDB$W_CNTRLFLGS field).

  It is therefore necessary to stall I/O requests for such a unit from the local host if the local host has lost quorum.

  If DUTU$REVALIDATE was called from EXE$CLUTRANIO and the disk is either dual-pathed or on a multihost controller, then the CLUTRAN flag is set in the UCB's STS field so as to remember this later.

- Routine EXE$MOUNTVER is called to start mount verification for each qualifying disk. EXE$MOUNTVER is passed a "dummy" IRP, namely the permanent IRP within the CDDB.

  — If EXE$MOUNTVER "reports" that it cannot start mount verification for this disk, then the following is done:

    o The VALID flag is cleared in the UCB. The unit is no longer software valid.

    o DUTU$TERMINATE_PENDING is called to terminate all I/O requests for this unit in the CDDB's restart queue, and all stalled requests for this unit queued to the UCB itself (IRPs in UCB$L_IOQFL queue). These requests are sent to I/O postprocessing with the condition value SS$_VOLINV (volume is software invalid).

— If EXE$MOUNTVER "reports" that it did start mount verification for the unit, then the following steps are performed if this controller is not undergoing initialization and the QUORLOST flag is not set:

  o The count of the number of UCBs linked to this CDDB waiting for mount verification to complete (WTUCBCTR field in CDDB) is incremented.

  o The address of the CDDB is stored in the UCB$L_WAIT_CDDB field.

NOTE

DUTU$REVALIDATE "knows" that EXE$MOUNTVER returns to its caller only if it successfully starts mount verification; EXE$MOUNTVER returns to its caller's caller if it fails to do so. Thus, immediately before calling EXE$MOUNTVER, DUTU$REVALIDATE pushes onto the stack the address of where EXE$MOUNTVER is to return in the event of failure.

Then, when EXE$MOUNTVER is actually called, the address of where to return within DUTU$REVALIDATE is also placed on the stack. So EXE$MOUNTVER will be "tricked" into returning to its caller in this case, regardless of whether it succeeds or fails.

The reason for doing this is as follows: EXE$MOUNTVER is called to start mount verification for a single disk. If it succeeds, it should return to its caller so that its caller can take appropriate action from there. If it fails, then it leaves the disk in a software invalid state and terminates pending I/O requests for that unit.

It has no reason for returning to its caller if it fails since it has resolved/terminated the request which led to its being called (and possibly others); so it returns to its caller's caller. However, DUTU$REVALIDATE needs to repeatedly call it from within a loop, once for each disk on a particular controller.

If called from DUTU$REVALIDATE, EXE$MOUNTVER must be "tricked" into returning to its caller regardless of whether it succeeds or fails.

• Once EXE$MOUNTVER has been called to start mount verification for all disks on the controller, DUTU$REVALIDATE is done. As disks complete mount verification, DUTU$MOUNTVER will receive control and perform "end of mount verification" processing.

4.5.10.2 EXE$MOUNTVER

This routine is called by DUTU$REVALIDATE as described above. But it is also called by IOC$ALTREQCOM when an end message is received with an MSCP status code indicating an error occurred while transferring data to or from a disk.

In both cases, EXE$MOUNTVER is passed the UCB for the unit on which mount verification is to be performed. If called by IOC$ALTREQCOM, it is also passed the IRP associated with the failed data transfer. If called by DUTU$REVALIDATE, it is passed a "dummy" IRP.

NOTE

EXE$MOUNTVER can handle situations where it is not passed an IRP. However, such situations do not arise with DUDRIVER.

Since DUDRIVER is concerned only with disks on "MSCP speaking" controllers, or VAXes emulating "MSCP speaking" controllers, a few special tests for non-MSCP disks are omitted from this flow.

- EXE$MOUNTVER begins by determining if mount verification is possible and necessary for the unit whose UCB it was passed.

  First, a necessary condition for mount verification to be performed on a disk device is that the unit be a file oriented random access device. In other words, either it is a disk, or "looks enough like a disk to be treated as a disk". For TMSCP tape volumes, the device must store CRCs and must not be at beginning of tape.

  However, these conditions by themselves is not sufficient.

  Combined with these first conditions, either of the following two cluster related situations is sufficient to invoke mount verification:

  — The local host is a member of a VAXcluster, but has lost quorum. In this case, the CLUTRAN flag in the UCB is unaltered; it remains set.

  — EXE$CLUTRANIO invoked mount verification, but quorum has not been lost by the local host. In this case, the CLUTRAN flag is cleared.

  If neither of the cluster related conditions is true, then all the following must be:

  — There must be a mounted volume in the unit.

  — The device must not be mounted foreign unless it is an TMSCP tape.

  — Mount verification must be enabled. (This is normally true by default when a file structured disk is mounted.)

  — The error which led to this routine being called must be one from which mount verification can recover.

    EXE$MOUNTVER was passed a VMS condition value. This condition value may be the I/O status associated with the IRP of a failed $QIO operation or the condition value may have been chosen by EXE$MOUNTVER's caller to intentionally force mount verification. This latter case applies when EXE$MOUNTVER is called by DUTU$REVALIDATE, which intentionally passes the condition value SS$_DEVOFFLINE.

    The condition value passed to EXE$MOUNTVER is looked up in a table of errors for which recovery is possible. This table is at location MVERR_TABLE, and contains the following:

Error Code	Description
SS$_MEDOFL	Media offline
SS$_VOLINV	Volume invalid
SS$_DEVOFFLINE	Device offline (because controller is inoperative)
SS$_SHACHASTA	Shadow set state/membership change
SS$_WRITLCK	Volume is writelocked

If the disk has been accidentally writelocked, then EXE$MOUNTVER branches to WRITLCK_HNDLR and does not return here. At WRITLCK_HNDLR, the IRP is merely recycled over and over again until the operator resolves the problem.

If a shadow set state change is involved, mount verification will still be performed. Shadow set mount verification is discussed in a separate book entitled *VMS Volume Shadowing Internals.*

The "media offline", "volume invalid", and "device offline" cases continue on with the next step described here.

- If the device is Served and if a *Server Mount Verification Routine* exists, JSB to that routine.
- The MNTVERIP flag in the STS field of the UCB is tested.

  If it is found to be set, then mount verification is already in progress for the disk. So this step is skipped, and no attempt is made to allocate a special mount verification IRP.

  If the MNTVERIP flag is clear, then it is set and this step is performed.

  An IRP is allocated from nonpaged pool. This IRP, called an "MVIRP", is to be used specifically for mount verification operations. The MVIRP flag is set in its IRP$W_STS field; this will allow the MVIRP to be processed by DUDRIVER **Even if Normal I/O is Stalled** for the unit.

  In the event of an allocation failure due to lack of nonpaged pool, a loop is provided to retry the allocation up to 10 times before giving up. EXE$MOUNTVER waits approximately 1 second between consecutive retries. If all the retries fail, this is "remembered" and handled later.
- If an IRP was passed to EXE$MOUNTVER by its caller, then routine DRIVER_CODE is called. DRIVER_CODE has the responsibility for branching to driver specific mount verification code, the address of which is kept in the Driver Dispatch Table at offset *DDT$L_MNTVER*. For DUDRIVER, this is routine DUTU$MOUNTVER.

  Since this is the first time DUTU$MOUNTVER is being entered for this disk, it in turn branches to the driver specific "begin mount verification" routine, *DUTU$BEGIN_MNTVER*. There, the following tasks are performed:

  — The MSCP_MNTVERIP flag in the UCB's DEVSTS field is tested to see if DUTU$BEGIN_MNTVER has been entered for this disk yet. In essence, a check is being made to see if those steps specific to the mount verification of a disk on an "MSCP speaking" controller have already begun for this unit.

    o If the flag is already set, then this routine has already been entered for this unit. So the UCB$W_RWAITCNT field is left unaltered.

    o If the flag is clear, it is now set and the UCB$W_RWAITCNT field is incremented. This is the first time this routine has been entered for this unit.

— If the IRP passed to EXE$MOUNTVER is not a "dummy" permanent IRP from the CDDB, but rather an IRP representing a "real I/O request", then it is inserted into the UCB's pending I/O request queue (UCB$L_IOQFL) in sequence number order.

This is done so that the IRP can be "remembered" and restarted after mount verification is completed.

— DUTU$BEGIN_MNTVER returns to DRIVER_CODE's caller. That would be EXE$MOUNTVER, which then continues on with the next step.

- Two steps ago, just prior to calling DRIVER_CODE, EXE$MOUNTVER tested the MNTVERIP flag. Based on the flag's setting, it decided whether or not to allocate an MVIRP.

  If the MNTVERIP flag was found already set, that step was skipped and no attempt was made to allocate an MVIRP since mount verification was already in progress for this unit. If that was the case, then EXE$MOUNTVER does not need to take further action. However, it does not want to report either "success" or "failure" to its caller. So EXE$MOUNTVER return's to its caller's caller.

NOTE

Remember, as described above, if EXE$MOUNTVER was called by routine DUTU$REVALIDATE, it is "tricked" into returning to its caller in this case.

If EXE$MOUNTVER did attempt to allocate an MVIRP but failed, even after the 10 retries, then it does not proceed any further. Instead, it branches to ERROR_EXIT where it handles the allocation failure in the following manner:

— Aborts mount verification for this unit.

— Notifies the operator that mount verification for this unit was aborted.

— Terminates all pending I/O requests for this unit with the error code SS$_VOLINV.

— The unit is left software invalid by having its VALID flag cleared to zero.

If EXE$MOUNTVER did allocate an MVIRP, then it calls routine *CALC_MVTIMEOUT* to calculate the mount verification timeout time for this unit. This is computed as the current time plus the number of seconds stored in location IOC$GW_MVTIMEOUT.

NOTE

IOC$GW_MVTIMEOUT contains the value of the sysgen parameter MVTIMEOUT. As of VMS V5.5, its value defaults to 3600 seconds (i.e. 60 minutes).

The mount verification timeout time is stored in the ASTPRM field of the MVIRP. (The ASTPRM field can be used this way here since MVIRPs don't use AST parameters.)

- If the UCB represents a shadow set virtual unit, then the code branches to PERFORM_SHADOW to perform shadow set mount verification recovery. Otherwise, the ordinary mount verification is performed by the instructions at PERFORM_VALIDATE.
- At PERFORM_VALIDATE, the two major tasks of mount verification are as follows:

  — First, routine PACKACK_VOLUME is called to set the unit online and make it accessible to the local host.

If this can be done using the current primary path controller, then it will be. If not, then an attempt is made to failover the unit to an alternate controller if the current primary controller is in a nonzero allocation class. A viable secondary path to the unit is sought.

- Routine DUTU$LOCATE_UNIT is called which in turn calls routine DUTU$BEGIN_CONN_WALK. This routine will determine if an existing secondary path is valid and if not will initiate a Connection Walk to locate a valid path.

  The steps performed by the Connection Walk routine are as follows:

  - Check to see if a Preferred Path has been specified by examining the UCB$L_PREF_CDDB field of the UCB.

    If a preferred path has been requested, the Load Balance flag in the CDRP is cleared and an attempt to validate the path is made. If the path is valid, it will be used. If the preferred path is not valid, a connection walk will be started to find a valid path.
  - Check to see if a Load Balance request has been made. If so, branch to perform connection walk.
- If the UCB already has a secondary path set up, then that path is used.
- If not, controllers in the same allocation as the current primary path controller are queried to see if they can access the disk until one says "yes", or until the list of controllers is exhausted. This process is known as connection walking.

If a secondary path is found, then it is made the new primary path, and the old primary path is made secondary. If not, then the original controller is used.

This task is done by issuing an IO$_PACKACK (pack acknowledge) to the unit. Doing so causes two MSCP commands to be issued to the controller. First, an ONLINE command is used to bring the unit online to the host. (This also brings it online to the controller if it isn't already.) If the ONLINE succeeds, then a GET UNIT STATUS command is used to ascertain the status and geometry of the unit.

PACKACK_VOLUME repeats the IO$_PACKACK approximately once a second until either it succeeds, or mount verification timeout occurs. Only if PACKACK_VOLUME succeeds does it return to its caller, PERFORM_VALIDATE, so that the next major task can be performed.

NOTE

The details of PACKACK_VOLUME are presented in the next section of this chapter. However, it is worth observing here that this repeating of the IO$_PACKACK operation will alternate between the two controllers to which a dual-path disk is ported.

— The second major task is to validate the volume on the unit; in other words, make sure that it is the same volume that was there before mount verification was needed. If the volume is mounted "foreign", then this second task is not performed; there would be no previous volume information with which to compare what is in the Homeblock and Storage Control Block.

If this unit has a foreign volume, then a branch is taken to the next step to wait for quorum to be regained. If the volume is not foreign, then this second task is done by calling routine VALIDATE_VOLUME. x>(VALIDATE_VOLUME)

- The Homeblock is read using an IO$_READPBLK operation. Then the following items are validated for the homeblock:
  - Checksum.
  - Volume Serial Number.
  - Volume Name.
- Next, the Storage Control Block (SCB) is read using another IO$_READPBLK operation. The following items from the SCB are validated:
  - Checksum.
  - Mount time.
  - Volume lock name (checked only if volume name didn't match above).

VALIDATE_VOLUME will now take one of four possible actions:

- If both the homeblock and SCB validate successfully, then VALIDATE_VOLUME returns to its caller, indicating "success".
- If previous information was not available with which to compare what is in the volume's homeblock and SCB, then VALIDATE_VOLUME aborts mount verification and leaves the VALID flag clear. The volume is not Software Valid. This, in turn, leads to the termination of all I/O requests in both the CDDB's CDRP restart queue, and the UCB's pending IRP queue with a Device Offline status.
- If wrong volume information is found in either the homeblock or the SCB, a message is issued to the operator. VALIDATE_VOLUME then loops back to its own beginning to try validating the volume again.
- For all other errors, VALIDATE_VOLUME will return an error code to its caller. This causes PERFORM_VALIDATE to loop back to its beginning and retry both of its major steps again (both PACKACK_VOLUME and VALIDATE_VOLUME).

NOTE

The test made by PACKACK_VOLUME to see if mount verification timeout has expired insures that PERFORM_VALIDATE won't loop within itself "forever".

The I/O done by PERFORM_VALIDATE will not be stalled, even though normal I/O is. This is because the IRPs used by PERFORM_VALIDATE have the flag set indicating they are MVIRPs.

- When both PACKACK_VOLUME and VALIDATE_VOLUME complete successfully, mount verification for the unit pauses until cluster quorum is regained if required.

  After quorum has been regained,

  — An operator message is sent indicating Mount Verification is Complete

  — The MVIRP is deallocated

  — The MNTVERIP and CLUTRAN flags are cleared. (The CLUTRAN may already have been cleared at the beginning of EXE$MOUNTVER.)

  — Routine DRIVER_CODE is called to resume I/O for this unit

- DRIVER_CODE branches to DUTU$MOUNTVER a second time. This time, DUTU$MOUNTVER branches to DUTU$END_MNTVER.
  - — The MSCP_MNTVERIP in the UCB's DEVSTS field is tested.

    If it is set (which it should be), then it is cleared and the RWAITCNT field in the UCB is decremented.

    NOTE

    This decrement of the RWAITCNT field matches the increment that was done by DUTU$BEGIN_MNTVER.

  - — The VALID flag in the UCB$L_STS field is tested to see if the unit has been declared software valid by mount verification.
    - o If the flag is clear, then the unit is software invalid. Routine DUTU$TERMINATE_PENDING is called to terminate all all CDRPs in the CDDB's restart queue, and all IRPs in the UCB's pending IRP queue (UCB$L_IOQFL).
    - o If the flag is set, then the unit is software valid. In this case, a check is made to see if Mount Verification was for the system disk unit and if we are a satellite node. If both of these conditions are met, the *PEM_BOOT Block* information is updated to indicate the new scssystemid of the path to the disk.

      Next, SCS$UNSTALLUCB is called to process the pending I/O requests.

      If the RWAITCNT field of the UCB is nonzero, SCS$UNSTALLUCB merely returns to its caller. If the RWAITCNT field is zero, then it starts up the stalled IRPs in the UCB$L_IOQFL queue by passing them to DUDRIVER's start I/O routine.

      NOTE

      The RWAITCNT would be nonzero if mount verification was invoked by the routine DUTU$REVALIDATE for all disks on a controller. Remember, DUTU$REVALIDATE was itself called either by DU$RE_SYNCH or by DU$CONNECT_ERR.

      Each of these routines intentionally increments the RWAITCNT field if they find it containing zero.This has the double effect of immediately stalling new IRPs being handed to DUDRIVER for this unit, and insuring that they are not unstalled until all CDRPs in the CDDB restart queue have been processed.

  - — As DUTU$END_MNTVER completes mount verification for each UCB, it decrements the WTUCBCTR field in the controller's CDDB. This is the count of the number of UCBs waiting for mount verification to complete. When this field becomes zero, DUTU$END_MNTVER checks to see if there are any CDRPs in the CDDB's restart queue. If there are, then it calls DUTU$RESTART_NEXT_CDRP to restart them. Details of routine DUTU$RESTART_NEXT_CDRP are presented in a later section of this chapter; The following is a summary:
    - o Processes any CDRPs in the CDDB's restart queue in "single stream" mode.
    - o Decrements the RWAITCNT field to zero.

- Calls SCS$UNSTALLUCB for each UCB linked to the CDDB to unstall any IRPs in the UCB's pending IRP queue.

4.5.10.3 PACKACK_VOLUME

This section follows the flow of an IO$_PACKACK function issued for a disk during mount verification.

- Determines if the IO$_PACKACK function is valid for the unit. It does so by verifying that the bit corresponding to IO$_PACKACK is set in the valid I/O function mask of the driver's FDT.

  NOTE

  If IO$_PACKACK is not a valid function for the unit, this routine merely turns into a NOP, simulates successful completion of the operation, and returns to its caller. This is not an issue here; IO$_PACKACK is valid for disks handled by DUDRIVER.

- Sets the VALID flag in the UCB. For now it is presuming success.
- Initializes the MVIRP passed to it by its caller.
  - — Standard SIZE, TYPE, and RMOD fields.
  - — Sets PHYSIO and MVIRP flags in IRP.
  - — IRP$W_FUNC field set to contain IO$_PACKACK.
- Calls EXE$MNTVERSIO to pass the IO$_PACKACK request to DUDRIVER.

  The details of EXE$MNTVERSIO and DUDRIVER's handling of the IO$_PACKACK MVIRP follow in the next section of this chapter.
- If the IO$_PACKACK function succeeds, then PACKACK_VOLUME merely returns to its caller.

  If the function fails but mount verification timeout has not expired, then PACKACK_ VOLUME loops back on itself and tries again after approximately a one-second pause.

  If mount verification timeout has expired, then mount verification for this unit is terminated and
  - — A "mount verification aborted" message is sent to the operator.
  - — The VALID flag is cleared in the UCB.
  - — Mount verification is disabled for this unit.

4.5.10.4 EXE$MNTVERSIO and Handling IO$_PACKACK MVIRP

- EXE$MNTVERSIO stores the address of a special I/O postprocessing routine, END_IO, in the PID field of the MVIRP.

  NOTE

  Eventually, when I/O postprocessing is invoked by an IPL$_IOPOST software interrupt, routine IOC$IOPOST will see that the PID field of the MVIRP does not actually contain a PID, but rather the address of a system space routine. This is because the high order bit of the PID field will be a "1". So, instead of performing normal I/O postprocessing, it will call the routine whose address is in the PID field. In this case, that routine will be END_IO.

  END_IO will copy into R0 and R1 the I/O status quadword from the MVIRP's IOST1 and IOST2 fields, and then return to EXE$MNTVERSIO's caller.

- Branches to *IOC$INITIATE* to pass the MVIRP to DUDRIVER's start I/O routine, DU_STARTIO.
  - — DU_STARTIO sees that normal I/O requests to the unit are stalled. Since this IRP is actually an MVIRP, it will branch to routine START_MOUNT_VER for special handling.

    START_MOUNT_VER selects the controller to which the IO$_PACKACK will be issued. This is where failover is triggered if it is necessary and possible.

    - o If a *Path Move* is in progress as denoted by the CDDB$V_PATHMOVE bit being set in the CDDB$W_STATUS field, the mount verification request will be completed immediately with a Device Offline error.
    - o The NOCONN flag in the CDDB for the controller which is currently providing the primary path to the disk is examined. If the flag is clear, then there is a connection with the controller's server, and this controller will be selected to receive the IO$_PACKACK. No failover will be attempted unless this PACKACK fails.
    - o If the flag is set, then there is no connection with the primary path controller's disk server. Routine DUTU$LOCATE_UNIT is called to see if there is a secondary path controller for the unit.
      - DUTU$LOCATE_UNIT first checks to see if *Load Balancing* has been requested.
      - DUTU$LOCATE_UNIT then calls routine DUTU$BEGIN_CONN_WALK to start a connection walk.
      - Routine DUTU$BEGIN_CONN_WALK performs the following steps:
        - — If a preferred path has been requested, clear the Load Balance flag in the CDRP and attempt to validate the path. If valid, the path will be used. If the preferred path is not valid, a connection walk will be started to find a valid path.
        - — Check to see if a Load Balance request has been made. If so, branch to perform connection walk.

- If the UCB does not reflect a secondary path, then the list of all CDDBs is scanned for controllers in the same allocation class as the primary path controller. To each such controller, a GET UNIT STATUS command is issued in an attempt to locate the disk on that controller. This scan continues until either a controller returns an end message indicating it already has the disk unit "online" (status = *MSCP$K_ST_SUCC*) or "available" (status = *MSCP$K_ST_AVLBL*), or until the list of CDDBs is exhausted. A controller returning an MSCP status code of "online" or "available" is a viable secondary path, even though this secondary path wasn't known until now.

  If DUTU$LOCATE_UNIT reports the presence of a secondary path, then

  o DUTU$MOVE_UNIT is called to alter the UCB linkages so that the secondary path is now the primary path, and what was the primary path is now the secondary path.

  o The new primary path controller is selected to receive the IO$_PACKACK. This is what will trigger failover.

  If DUTU$LOCATE_UNIT fails to find a secondary path, then the MVIRP is passed to IOC$ALTREQCOM with the condition value SS$_DEVOFFLINE. From there, it goes to I/O postprocessing, which then passes it to the code at END_IO for the reasons explained above.

— Given that START_MOUNT_VER was able to select a controller to which an IO$_PACKACK can be sent, it then branches back into the mainstream of routine DU_STARTIO. From there a standard dispatch is made to START_PACKACK based on the function code IO$_PACKACK in the IRP$W_FUNC field (which is the same as the CDRP$W_FUNC field).

- START_PACKACK actually performs the IO$_PACKACK function, which consists of issuing an MSCP ONLINE command followed by an MSCP GET UNIT STATUS command.

  — In general, if the end message corresponding to the ONLINE command returns a status code other than "success" (MSCP$K_ST_SUCC), then the IO$_PACKACK operation is terminated with an appropriate VMS condition value. The choice of the condition value depends on the MSCP status returned in the end message. In this case, the GET UNIT STATUS will not even be issued.

  A particularly important exception is when the MSCP status in the end message is "offline" (MSCP$K_ST_OFFLN) and the sub-status is either *Unknown Unit* or *Inoperative Unit*. Then DUTU$LOCATE_UNIT is called to locate another path to the unit. If one is found, DUTU$MOVE_UNIT is called to make the new path primary, and the old primary path secondary. Then the ONLINE is retried using the new primary path.

  This exception (in particular, the "inoperative" sub-status) addresses the situation wherein the controller associated with the CDDB passed to START_PACKACK is still alive, the disk class driver still has a connection with the controller's disk server, but the controller itself cannot communicate with the unit. This facilitates failover in situations such as when an SDI on an HSC fails, but the HSC survives and can't access the disks on that SDI.

  There is a retry limit hard coded in START_PACKACK for this exception. The value for VMS V5.5 is four retry attempts.

— If the ONLINE succeeds, but the GET UNIT STATUS fails, then the drive is assumed to have gone offline, and the code branches back to the beginning of START_PACKACK to retry both the ONLINE and GET UNIT STATUS commands.

— If both operations succeed, *RECORD_ONLINE* and *RECORD_UNIT_STATUS* are called to store information in the UCB. Some of this information includes media ID, unit size, volume serial number, and geometry of the disk. The condition value returned will be SS$_NORMAL, and the VALID flag is left set in the UCB.

NOTE

Upon entry to START_PACKACK, the RWAITCNT field in the UCB is incremented to stall new IRPs being handed to DUDRIVER for this unit. When the IO$_PACKACK operation completes, this field is decremented. Within the context of mount verification, this field will still not be zero; so the stalled IRPs remain stalled. However, if the IO$_PACKACK was requested in some other context, then the RWAITCNT field may be zero now. So SCS$UNSTALLUCB is called to test this field and, if it is zero, unstall any IRPs in the UCB's pending IRP queue (UCB$L_IOQFL).

4.5.10.5 Restarting CDRPs

Routine DUTU$RESTART_NEXT_CDRP is responsible for restarting I/O requests in a CDDB's CDRP restart queue after mount verification completes for all units on a controller. DUTU$END_SINGLE_STREAM restarts I/O requests in the pending IRP queue of each UCB attached to that CDDB after single stream processing of CDRPs is complete.

4.5.10.5.1 DUTU$RESTART_NEXT_CDRP and DUTU$END_SINGLE_STREAM

- Attempts to remove the next CDRP from the CDDB's restart queue.
  — If there are no more CDRPs in the restart queue, then DUTU$RESTART_NEXT_CDRP branches to DUTU$END_SINGLE_STREAM where the following is done:
    - The SNGLSTRM flag is cleared in the CDDB.
    - Loops through all UCBs handled by this controller, resuming IRPs in each UCB's I/O wait queue (UCB$L_IOQFL) if there is no other reason I/O is stalled on the unit.

      This is done by decrementing the UCB's RWAITCNT field, and then calling routine SCS$UNSTALLUCB to remove IRPs from the pending IRP queue and submit them to the driver's start I/O routine, DU_STARTIO.
    - Clears the RECONNECT flag in the CDDB.
    - Returns to its caller, since single stream mode is now complete for this controller.

— If a CDRP was fetched from the restart queue, then:
  - The SNGLSTRM flag in the CDDB is set if this is the first time routine DUTU$RESTART_NEXT_CDRP was called for this CDDB. (This indicates that single stream processing is being done for CDRPs queued to the CDDB's restart queue.)
  - A branch is made to DU_RESTARTIO with the CDRP.

    From this point on, the processing of the CDRP proceeds just as if it were being seen for the first time. An MSCP command is built and sent to the controller, and the request is suspended, pending receipt of the corresponding end message.

- When the end message is received, the request resumes immediately after the SEND_MSCP_MSG macro.
  — As explained in the detailed flow for the normal read or write request,
    - DUDRIVER verifies that there was no MSCP error reported in the end message STATUS field. (If there is, and this was a read or write, then a branch is made to TRANSFER_MSCP_ERROR, where the error is handled in the normal manner.)
    - Quadword IOSB information is constructed based only on the segment just completed.
    - DUTU$DEALLOC_ALL is called to release SCS resources held by the CDRP.
  — Instead of automatically branching to IOC$ALTREQCOM, DUDRIVER sees that the SNGLSTRM flag is set in the CDDB, indicating that single stream CDRP processing is in progress.
    - If the CDRP is for mount verification, then it is allowed to pass to IOC$ALTREQCOM through a JMP.
    - Otherwise, the code completes the request by a JSB to IOC$ALTREQCOM so that control can be regained. The code then branches back to DUTU$RESTART_NEXT_CDRP to pick up the next CDRP in the CDDB's I/O restart queue and process it.

4.5.10.5.2 Preventing an Infinite Loop

If single stream processing was in progress when the connection unexpectedly failed, it is possible that the CDRP at the head of the restart queue is the CDRP being processed at the time of the connection failure. This situation could lead to an infinite loop if there were no mechanism in place to prevent it. The following prevents this infinite loop potential.

Within the RSTRTCDRP field of the CDDB is stored the address of the CDRP last fetched from the CDRP restart queue. If the address of the CDRP just fetched from the restart queue is different from what is currently in the RSTRTCDRP field, then:

- The RSTRTCDRP field is reset to contain the address of the newly fetched CDRP.
- A retry count field, RETRYCNT, in the CDDB is initialized to the value of the parameter MAX_RETRY.
- The CDRP is handled as described above.

If the address of the CDRP matches the content of the RSTRTCDRP field, then DUTU$RESTART_NEXT_CDRP is confronted with this special case. It therefore does the following:

- Decrements the RETRYCNT field in the CDDB.
- If the RETRYCNT field goes to zero, then the CDRP is not retried again; it is submitted to I/O postprocessing with the error *SS$_CTRLERR* to be returned to the process in its I/O status block.
- If the RETRYCNT field is still nonzero, then the CDRP is retried as described above.

As of VMS V5.5, the value of MAX_RETRY is hard coded at the beginning of DUDRIVER to be 2.

4.6 DUDRIVER BUGCHECKs for Non-shadowed Disks

The following table summarizes the four *BUGCHECKs* that can occur in the disk class driver when dealing with non-shadowed disk units. All of these, and more, can occur with shadow set virtual units as well. BUGCHECKs associated with shadow sets are covered in a later chapter.

BUGCHECK	Description and Sources
DISKCLASS	DDB passed to controller initialization routine, DUCONTROLLERINIT, has more than one UCB attached to it.
	The end message from the SLUN RSVP request is found to be inconsistent: The MSCP unit number is still the SLUN RSVP unit or the SLUN bit is not set.
	An MSCP server has returned invalid geometry information in the end message for a GET UNIT STATUS command for an online unit.
	An attempt to send an MSCP message in the DU$TMR routine has returned with a CDT in invalid state error.
	An UNSOLICITED interrupt has been received.
	Inconsistency in HIRT or RCT database.
INCONSTATE	Routine DUTU$CREATE_CDDB thought it was working with the UCB of the boot device, then discovered it wasn't.
	While creating a secondary path linkage for a UCB, another UCB with a duplicate unit number was discovered.
	While validating that the DDB returned from routine DUTU$FIND_DDB is the correct one for the current CDDB, a systemid for a nonexistent System Block is found in the CDDB.

BUGCHECK	Description and Sources
MSCPCLASS	The MSCP server has sent a poisoned end message requesting the bugcheck.
	When creating a CDDB, an invalid controller letter was found in the CDDB
	When creating a CDDB for a newly discovered controller, another CDDB with the same system id and UCBs already attached was found in the I/O database.
	The CDDB$V_DISABLED bit is set in the CDDB$W_STATUS word for the System Device indicating that the system device is on a disabled controller.
	The operating system has been up at least 45 seconds, but the system device has not yet been found by unit polling.
	The Original UCB (system device) is still in an initing state after returning from the fork thread that should have linked it in.
	"Doubly waiting" UCB. DUTU$REVALIDATE has started mount verification for a unit on a controller, but then finds that the UCB indicates that some other controller is waiting for mount verification to complete for the unit.
	DUTU$BEGIN_MNTVER has detected that an attempt is being made to start mount verification for a unit which is already in mount verification.
	Routine DUTU$LOCATE_UNIT has received the reserved SLUN RSVP unit number from an VMS MSCP server or the SLUN bit was not set in the end message.
	Routine DUTU$NEW_UNIT has found an inconsistency in an end message returned from a VMS MSCP server. (SLUN bit not set, SLUN RSVP received, invalid controller letter, invalid unit number)
	Routine DUTU$NEW_UNIT has found a previously known nonemulated disk unit requesting that the MSCPUNIT be recalculated (SLUN RSVP set).
	RWAITCNT field in new UCB is not zero at the end of setting up new linkage.
	Routine DUTU$SETUP_DUAL_PATH has found an empty controller mask in the CDDB.
	An access path attention message has been received asking to failover a disk, but the primary path is a nonemulated path.
	Routine DUTU$FIND_DDB was called to find a DDB for a generic device name (e.g. DUA, DJA, ...) without a proper fork block.

BUGCHECK	Description and Sources
	An attempt was made to insert something other than a CDRP into a CDDB's CDRP restart queue. An attempt was made to insert something other than an IRP/CDRP pair into the I/O postprocessing queue. A UCB's RWAITCNT field was found to contain an invalid quantity. When logging the receipt of an invalid MSCP command end message status code, a co-routine has improperly attempted to return to its caller due to a problem.
RESEHX	Unable to allocate an IRP to do an IO$_PACKACK for the boot device.

Chapter 5

The VMS Based MSCP Server

5.1 Introduction

The role of the *VMS based MSCP Disk Server* is to implement *Mass Storage Control Protocol* (MSCP) on a VMS system. This allows the VMS system to emulate a DSA controller, and in so doing, make one or more of its local MASSBUS, UNIBUS, or DSA disks available to other hosts in a VAXcluster.

This mechanism also allows VMS members with a direct connection to a remote disk server (such as a *Hierarchical Storage Controller* (HSC) or *Integrated Storage Element* (ISE)) to provide access to its disks to VAXcluster members that do not have direct connections (ie: NI members). This process requires that the *Allocation Class* of the VMS system be the same as that of the remote MSCP server.

5.2 MSCP Disk Serving

The MSCP server software must be loaded and started at boot time by setting the *MSCP_LOAD* Sysgen Parameter. The ability to load the server software interactively from the Sysgen Utility is no longer supported in VMS V5.x. The Sysgen Utility will return the following error if an attempt is made to dynamically load the server:

"%SYSGEN-E-CMDOBS, MSCP server must be loaded by setting SYSBOOT parameter MSCP_LOAD"

The Sysgen Parameter *MSCP_LOAD* is used to direct the *Standalone Configure Process* (STACONFIG) to load the MSCP server software at boot time. This parameter can be set to the values listed in Table 5–1.

Table 5–1: Sysgen Parameter MSCP_LOAD settings

Value	Description
0	Do not load the MSCP server. This is the default value.
1	Load the MSCP server and serve disks as specified by the MSCP_SERVE_ALL parameter.
>1	Load the MSCP server and assign the provided value as the server's *Load Capacity*

Once the MSCP server has been loaded and started, local disks, HSC disks, and ISE disks can be either *Automatically* or *Selectively* made available to other VMS hosts in the VAXcluster.

5.2.1 Automatic Disk Serving

The Sysgen Parameter *MSCP_SERVE_ALL* can be used to automatically serve disks at system startup. This parameter can be set to the values listed in Table 5–2.

Table 5–2: Sysgen Parameter MSCP_SERVE_ALL settings

Value	Description
0	Do not serve any disks. This is the default.
1	Serve all available disks.
2	Serve only locally-attached (non-HSC) disks.

5.2.2 Selective Disk Serving

Disks can individually be selected for MSCP serving as well. This would be accomplished by setting the MSCP_SERVE_ALL parameter to zero and issueing the following DCL command for each disk to be served:
$SET DEVICE/SERVED {disk:}

The MSCP server software must be loaded prior to setting any disk served.

5.2.3 Dual Ported Disks

If the local disk being "served" to the cluster is a *MASSBUS disk* which is also *Dual-Ported* to another VAX, then the /DUAL_PORTED qualifier should be included in this DCL command:
$SET DEVICE/SERVED/DUAL_PORTED {disk:}

This DCL command should be issued on BOTH VAXes to which the disk is dual-ported. This allows remote hosts to see that the disk as dual-pathed.

NOTE

A disk should be set served only on the host(s) to which it is ported.

5.2.4 The MSCP Server

Once the local VAX has loaded and started the MSCP server, remote VAXes can issue MSCP commands to the local VAX in the same way as they would to an HSC or ISE. The local VAX is acting as a *Logical Controller* for any of its local disks or HSC disks which it has set served. The VMS based MSCP server engages in MSCP dialogue with remote *Disk Class Drivers* in the same way as would the MSCP server in an HSC or ISE.

When the local VAX receives an MSCP command from a remote VAX, its SCS layer routes the command to the message input routine within the MSCP server software. The message input routine then dispatches the command to an appropriate handler.

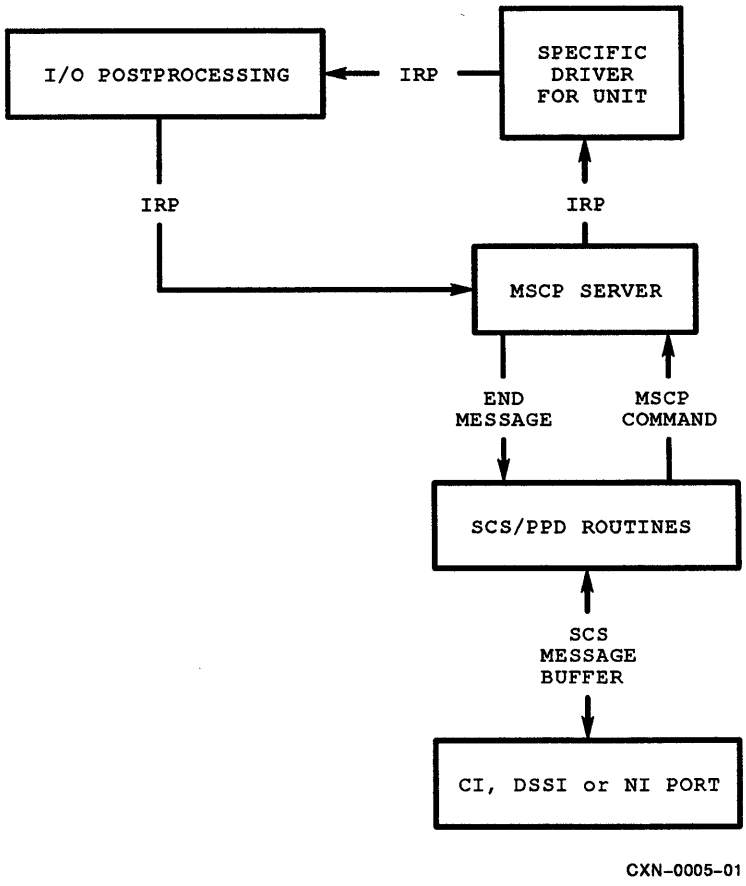
If the command requires no actual interaction with a disk unit, (such as a request for the current unit status), then the command is handled entirely within the MSCP server. An end message containing an MSCP status code and appropriate information is sent to the requesting remote host.

If the command requires interaction with a disk, (such as a read or write operation), the MSCP server allocates and initializes an IRP to represent the request contained within the MSCP command. The IRP is then passed to the driver for the unit, just as if the request had originated on the local host. Upon completion of the operation represented by the IRP, I/O postprocessing returns the IRP to the server software. If necessary, the server may adjust the IRP due to segmentation of the request, and resubmit it to the driver.

If a segment of data is to be written to the disk, it is fetched from the remote host before submitting the IRP to the driver. If a segment of data was read from the disk, it is sent to the remote host before the IRP is adjusted and resubmitted to the driver to read the next segment. An end message is sent to the remote host's disk class driver after the server has completed the entire request defined by the MSCP command, or upon premature termination of the request for any reason (such as an error).

Figure 5–1 illustrates the basic flow through the server:

Figure 5–1: MSCP Server Flow



5.3 MSCP Server Database and Initialization

To understand the internals of the VMS based MSCP server, it is essential to be familiar with the data structures the server maintains as it emulates a DSA controller. The next few sections of this book introduce these data structures, present some of the most important information contained within them and how they are linked together, and then surveys the steps in their initialization.

5.3.1 MSCP Server Data Structures

There are five major data structures used by the VMS based MSCP disk server to maintain the information it needs to service MSCP commands from remote hosts. These are listed in Table 5–3:

Table 5–3: MSCP Server Data Structures

Structure	Name
DSRV	Disk Server Structure
UQB	Unit Queue Block
HQB	Host Queue Block
HULB	Host Unit Load Block
HRB	Host Request Block

5.3.1.1 HRB - Host Request Block

Whenever the MSCP server's SCS message input routine, MSG_IN, receives an MSCP command from a remote host, the first major step it takes is to allocate a *Host Request Block* (HRB). This data structure represents the context of the request contained within the command. Some of the items of information maintained in an HRB are:

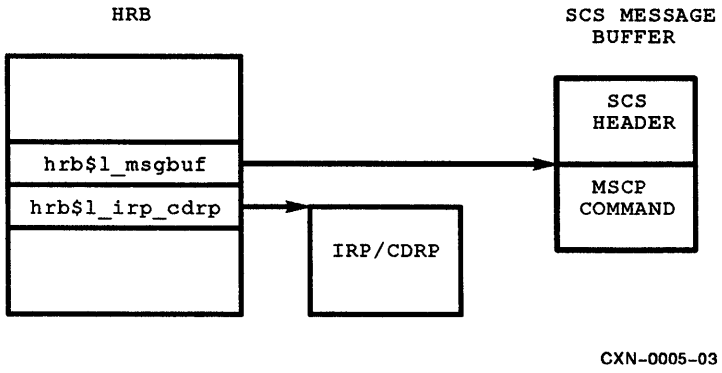
- State of the request (e.g. waiting for SCS credit, mapping a buffer, queued to a driver, sending or receiving data, sending a message to a host, etc.).
- Command status (decremented at various points in the processing to indicate that progress has been made by the server on the command represented by the HRB).
- Status flags (e.g. aborted, end message needs to be sent, map resources allocated to the command, etc.)
- Descriptor address, starting address, and length of a buffer allocated to handle data transfers if any are associated with the command.
- Quantities typically associated with a data transfer, if one is involved.
  - LBN Logical Block Number.
  - OBCNT Original Byte Count.
  - ABCNT Accumulated Byte Count.
  - SVAPTE System Virtual Address of Page Table Entry for local buffer.
  - BOFF Buffer Offset.
  - BCNT Actual Byte Count for a segment.

MSCP protocol dictates that for every command received, an end message must be sent to the the host which issued the command. The VMS based MSCP disk server does this by altering selected fields in the received command, and depositing information in others. It then turns the command around and sends it back. Thus, the address of the received command is also stored in the HRB.

Any request to transmit a message via SCS services requires that a CDRP be allocated and initialized to describe the request to the SCS layer. Furthermore, most MSCP commands received by the server involve requests from remote hosts to transfer data.

This makes it necessary to allocate and initialize an IRP to describe the transfer, and then queue the IRP to the particular driver for the disk unit involved in the transfer. For both of these reasons, an IRP/CDRP pair is allocated along with the HRB, and the address of the IRP/CDRP pair is also stored in the HRB. Figure 5–2 illustrates some of the fields in the HRB:

Figure 5–2: HRB fields



5.3.1.2 HQB - Host Queue Block

Before passing the Host Request Block to routines which process the command represented by the HRB, MSG_IN must first associate the HRB with the remote host that sent the command. This is done by inserting the HRB into a queue maintained in another data structure known as a *Host Queue Block* (HQB).

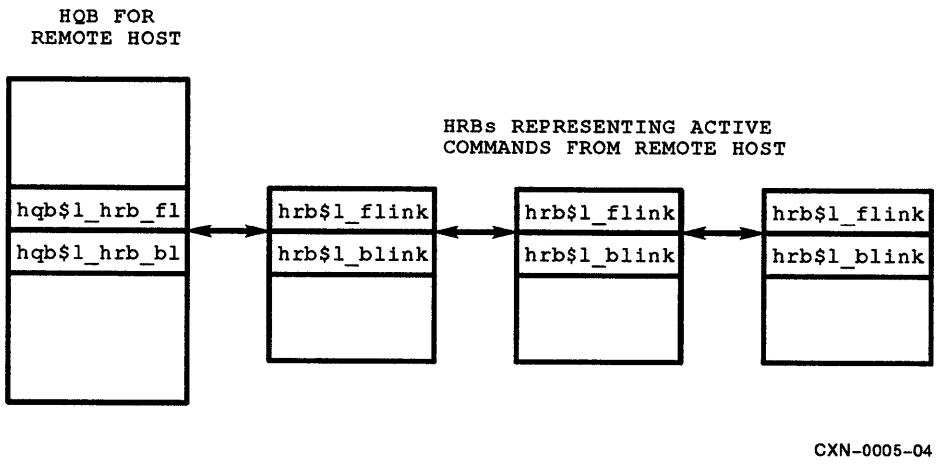
The MSCP server maintains an HQB for each remote host with which it has an SCS connection. (Actually, the connection is between the local host's MSCP server and the remote host's disk class driver.) An HQB contains information that pertains to its associated host, such as

- Address of the CDT describing the SCS connection with the remote host's disk class driver.
- Host settable controller flags.
- Host timeout interval.
- Time the remote host issued a SET CONTROLLER CHARACTERISTICS command.

- Number of active requests (i.e. HRBs queued to the HQB).
- A pointer to a vector of addresses representing Host Unit Load Balance structures describing I/O load information (See Section 5.3.1.4).

Figure 5–3 illustrates the relationship of the HRB to the HQB.

Figure 5–3: HRB relationship to the Host Queue Block



A Host Queue Block is allocated from nonpaged pool and initialized when the local MSCP server ACCEPTs an incoming SCS CONNECT request from a remote host's disk class driver. If the SCS connection fails, then the HQB is deallocated.

5.3.1.3 UQB - Unit Queue Block

The local MSCP server maintains one *Unit Queue Block* (UQB) for each disk which has been made available to remote hosts through the local VMS MSCP Server. A *Server Local Unit Number* (SLUN) is assigned to index each UQB. The following are some of the items of information contained within a UQB:

- State of the unit.
  - — AVAILABLE
  - — ONLINE
  - — OFFLINE
- Flags.
  - — SEQ - sequential command is executing for this unit
  - — WRTPH - unit is hardware write protected
  - — WRTPS - unit software write protected by use of /NOWRITE qualifier when volume mounted

- Unit ID. (Allocation Class, Controller letter, Unit Number etc.)
- Address of UCB for the unit associated with this UQB.
- Queue of "blocked" HRBs, waiting for sequential command to complete.
- Server Local Unit Number

5.3.1.4 HULB - Host Unit Load Block

The local MSCP server maintains one *Host Unit Load Block* (HULB) for each unique VMS host and disk unit combination whose served I/O requests are being handled by the local VMS system.

The HULB maintains information related to the I/O load being impressed on a particular unit by a specific host. These statistics are used to distribute the MSCP serving workload equitably amongst all available VMS based MSCP servers for a device when a new Disk Class Driver to VMS based MSCP server connection is formed.

The HULB maintains an *Operation Count* field to account for all I/O activity from a particular host to a given unit. The Host Number and Unit Number are maintained in the structure as well to identify the combination.

5.3.1.5 DSRV - Disk Server Structure

The *Disk Server Structure* (DSRV) is the principle data structure dealing with MSCP Server emulation. It contains the listheads for the HQB, the UQB and the HULB data structures as well as information about the servers *Load Capacity*, buffer management information, and miscellaneous other information to track its connections with remote disk class drivers. The DSRV also maintains a vector of pointers to each unique UQB. These are used to speed access to the Unit information in the UQBs by providing a *Server Local Unit Number* (SLUN) which is an index into this list. Access to a particular unit can then be made by using its SLUN to identify it.

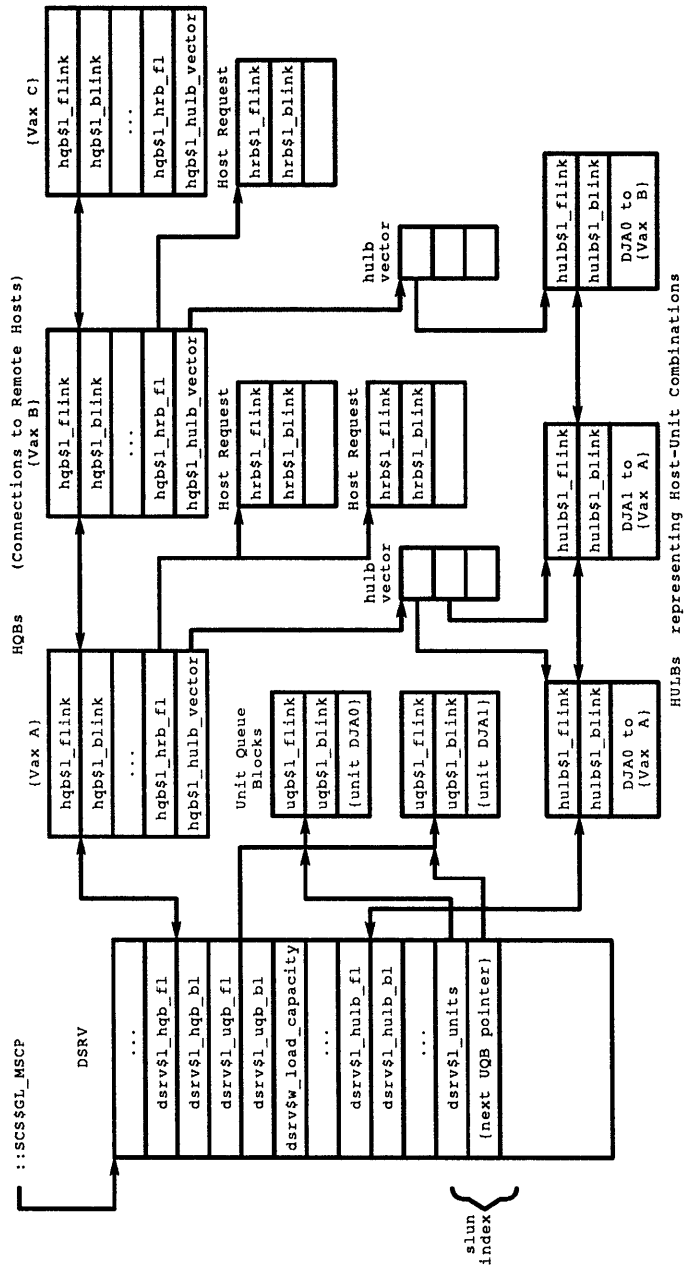
Some of the data items kept in the DSRV are as follows:

- Information describing a pool of transfer buffers allocated by the server to act as temporary holding areas for data being exchanged between a remote host and a local disk. Section 5.4.4 provides details of this.)
- Controller information such as controller flags and the controller timeout interval maintained by any DSA controller.
- Number of disks being served.
- Number of remote disk class drivers with which SCS connections are open.
- Load Capacity Information
- HQB, HULB and UQB listheads.
- Two tables of statistics.
  - — Counters for all MSCP op codes received by the MSCP server since it was loaded.

  - — Counters for different size block transfers handled by the MSCP server since it was loaded.
- Vector of Unit Queue Block addresses

The DSRV is initialized during MSCP server startup. Its address is kept in global location *SCS$GL_MSCP*. Figure 5–4 illustrates the format of the DSRV structure and its relationship to the other server data structures:

Figure 5–4: Disk Server Structure Layout



CXN-0005-05

5.4 MSCP Unit Numbers and Identifiers

MSCP Unit Numbers and Identifiers are used to uniquely identify disk units within a VAXcluster environment.

Device names are typically of the form *ddcu*, where **dd** denotes the device type, **c** is a controller designation, and **u** is a unit number. The VMS disk class driver constructs disk device names based on the MSCP media identification and MSCP unit number returned in the END MESSAGE corresponding to a GET UNIT STATUS command.

5.4.1 MSCP Media Identification

When a disk class driver establishes a connection with an MSCP server, the driver polls the server to determine what disks are available via that server. This is done by a series of GET UNIT STATUS commands, each with the "next unit" flag set (See Section 2.4.3 for detail).

In response to each such command, the server responds with an End Message containing the MSCP unit number of a disk. All subsequent commands and associated End Messages specifically related to the disk unit include the unit's *MSCP Unit Number*.

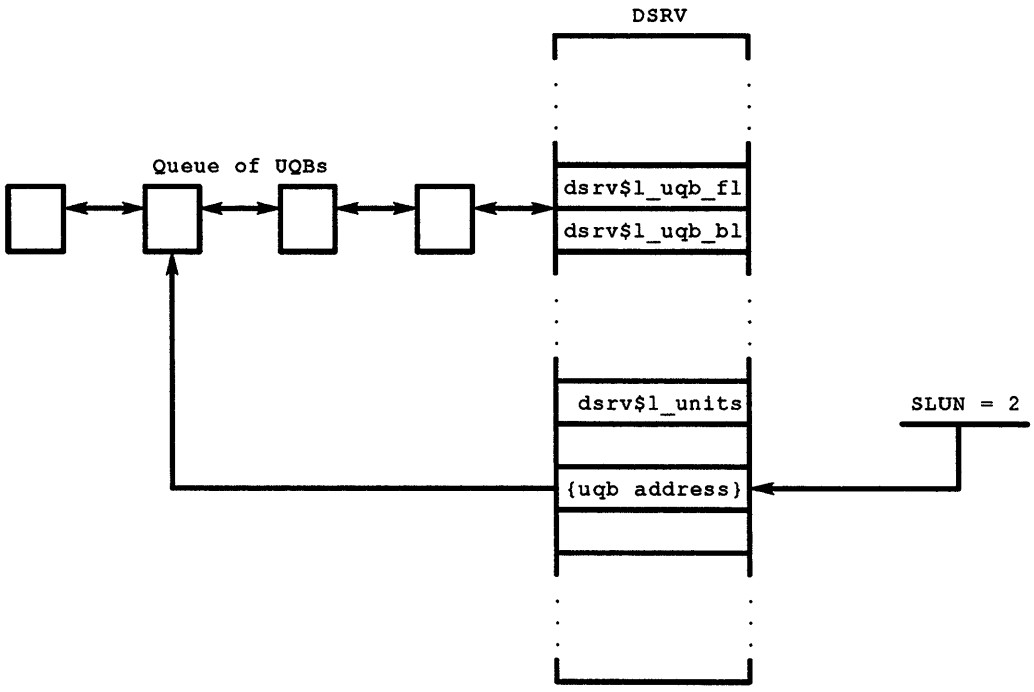
The VMS based MSCP server maintains UQBs representing its served disks in a queue, the head of which is in the DSRV at offset DSRV$L_UQB_FL. When a command related to a served disk is received from a disk class driver, the appropriate UQB must be located in the list.

To facilitate this search, the DSRV maintains a list at offset *DSRV$L_UNITS* called the *DSRV Unit Table* with the addresses of all the UQBs in the queue.

A Server Local Unit Number is assigned to each Unit. The SLUN is an index into the unit table starting with index zero. Location DSRV$W_NUM_UNIT stores the number of entries in the list.

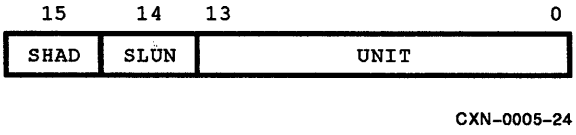
Figure 5–5 illustrates the queue of UQBs present for VMS based MSCP servers, and the SLUN which was introduced in Version 5.0.

Figure 5–5: Server Local Unit Number



MSCP commands and End Messages exchanged between a VMS based MSCP server and a disk class driver contain the following format MSCP unit number for VMS V5.x systems. This format involves a disk unit's SLUN and is illustrated in Figure 5–6. VMS versions prior to V5.0 did not implement the DSRV Unit Table and subsequently did not utilize the SLUN.

Figure 5–6: MSCP Unit Number for the VMS based MSCP Server



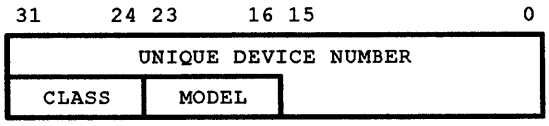
Field	Description
SHAD	Known as the MSCP$V_SHADOW flag, this bit is set in MSCP unit numbers returned by HSCs for shadow set virtual units. Otherwise it is clear, even in MSCP unit numbers returned for virtual units by the VMS–based MSCP server.
SLUN	This flag, known as the MSCP$V_SLUN flag, is set in MSCP unit numbers returned by a VMS–based MSCP server. Otherwise, it is clear.
UNIT	This field contains the actual drive unit number if the server resides in a DSA controller. But if the server is VMS–based, this field contains the DSRV unit table index identifying the UQB associated with the disk unit.

The MSCP server anticipates that it may have to process commands from a Version 4.7 disk class driver. While it maintains the newer *SLUN format* MSCP unit number for each disk unit at offset *UQB$W_SLUN* in that unit's UQB, it also keeps the Version 4.7 format MSCP unit number in the UQB at offset *UQB$W_OLD_UNIT*.

5.4.2 Unit Identifier

MSCP requires each controller and unit to have a unique 64–bit identifier. VMS V4.7 through V5.5 maintain a controller identifier at offset *CDDB$Q_CNTRLID* in a CDDB, and a unit identifier at offset *UCB$Q_UNIT_ID*. The general format defined by the MSCP specification for both controller and unit identifiers is shown in Figure 5–7.

Figure 5–7: 64 bit Unique Identifiers



Field	Description
CLASS	This field indicates a generic subsystem category, such as controller or disk unit. Values defined for this field by the MSCP specification are provided in the first of the next four tables.

Field	Description
MODEL	This field indicates a specific model within the subsystem class specified by the CLASS field. Controller model values and disk model values are presented in the last three of the next four tables.
UNIQUE DEVICE NUMBER	This field is intended to uniquely identify the device from among all devices defined by the CLASS and MODEL fields. The MSCP specification suggests, but does not require that a device serial number be used as a UNIQUE DEVICE NUMBER.

The next table summarizes the CLASS values supported by VMS Versions 4.7 and 5.0.

CLASS	Value (decimal)	Description
MSCP$K_CL_CNTRL	1	MSCP Controller
MSCP$K_CL_DISK	2	Disk Class Device (DEC Standard 166)
MSCP$K_CL_TAPE	3	Tape Class Device
MSCP$K_CL_D144	4	Disk Class Device (DEC Standard 144)
MSCP$K_CL_LDR	5	Media Loader

If the CLASS field indicates the identifier is for an MSCP controller, then the table below summarizes the MODEL fields that pertain to disk controllers.

MODEL	Value (decimal)	Comment (if any)
MSCP$K_CM_HSC50	1	
MSCP$K_CM_UDA50	2	
MSCP$K_CM_RC25	3	AZTEC integrated controller
MSCP$K_CM_EMULA	4	VMS–based MSCP server (software)
MSCP$K_CM_UDA50A	6	also sometimes known as the UDA52
MSCP$K_CM_RDRX	7	RQDX1 and RQDX2 controller
MSCP$K_CM_RUX50	10	
MSCP$K_CM_KDA50	13	
MSCP$K_CM_RV20	15	integrated controller
MSCP$K_CM_RRD50	16	
MSCP$K_CM_RRD50Q	16	
MSCP$K_CM_KDB50	18	
MSCP$K_CM_RQDX3	19	
MSCP$K_CM_RQDX4	20	
MSCP$K_CM_DSSI_ DISK	21	
MSCP$K_CM_RRD50U	26	
MSCP$K_CM_KDM70	27	
MSCP$K_CM_HSC70	32	
MSCP$K_CM_HSC40	33	
MSCP$K_CM_HSC60	34	
MSCP$K_CM_HSC90	35	
MSCP$K_CM_RF30	96	
MSCP$K_CM_RF71	97	
MSCP$K_CM_RF31	100	
MSCP$K_CM_RF72	101	
MSCP$K_CM_RF73	102	

When the disk class driver establishes a connection with an MSCP server in a DSA controller, or a VMS based MSCP server in a remote VAX, it issues a SET CONTROLLER CHARACTERISTICS command to the server. The *64 bit controller identifier* is returned by the server in the End Message corresponding to that command.

The disk class driver stores the controller identifier in the CDDB at offset CDDB$Q_ CNTRLID. Since the VMS based MSCP server is emulating a DSA controller, it maintains its "emulated" controller identifier in the DSRV at offset *DSRV$Q_CTRL_ID.*

5.4.2.1 MSCP Class Number

The *Class* and *Model* fields in controller identifiers for DSA controllers and the VMS based MSCP server are consistent with what has been described. The Class is set to 1, indicating an MSCP controller.

5.4.2.2 MSCP Model Number

The Model field then indicates the controller model such as 35 for HSC90 or 27 for KDM70; and, of course, the controller model for the VMS based MSCP server is 4 indicating that it is an "emulated controller".

5.4.2.3 MSCP Unique Device Number

There is some variation in the *Unique Device Number*. With local DSA controllers, this field contains the Controller Serial Number. With HSCs this field contains the low order 48 bits of the HSC's ID parameter (its SCS System ID). And with a VMS based MSCP server, the Unique Device Number is the low order 48 bits of the SCSSYSTEMID parameter for the VAX in which the server resides.

The next table summarizes the DEC Standard 166 (MSCP) values for the MODEL field when the CLASS field contains a "2". VMS does not have symbols which equate to these model numbers; so only their numerical values are shown here.

MODEL (decimal)	Disk Device	Description
1	RA80	121 MB, 14", fixed
2	RC25	26 MB, 8", removable
3	RCF25	26 MB, 8", fixed
4	RA60	205 MB, 14", removable
5	RA81	456 MB, 14", fixed
6	RD51	10 MB, 5.25", fixed, full height
7	RX50	400 KB, 5.25", single–sided 96 TPI floppy, full height, dual drives (800 KB total)
8	RD52	33 MB, 5.25", fixed, full height
9	RD53	71 MB, 5.25", fixed, full height
10	RX33	1200 KB, 5.25", double–sided 96 TPI floppy, half height
11	RA82	622 MB, 14", fixed
12	RD31	20 MB, 5.25", fixed, half height
13	RD54	160 MB, 5.25", fixed, full height
14	RRD50	500 MB, 4.75", removable, DAD (optical format)

MODEL (decimal)	Disk Device	Description
15	RD32	40 MB, 5.25", fixed, half height
17	RX18	180 KB, 5.25", single–sided, 96 TPI floppy, full height
18	RA70	280 MB, 5.25", fixed, full height
19	RA90	1.216 GB, 9", fixed
24	RD33	80 MB, 5.25", fixed, half height

When a VMS disk class driver establishes a connection with an MSCP server, it polls the server for disk units by means of GET UNIT STATUS (GUS) commands. GET UNIT STATUS commands are also used by a disk class driver when a disk unit enters mount verification. If the server resides in a DSA controller, then the above DEC Standard 166 model numbers are returned by the server as part of the 64–bit unit identifier in each End Message corresponding to a GUS command.

Normally, an MSCP speaking controller will return a serial number as the Unique Device Number for a disk unit. When the unit does not have an intrinsic serial number, the controller returns the Unit Number instead. For example, an RQDX3 would return "1" as the UNIQUE DEVICE NUMBER for a RD54 known as DUA1.

VMS has symbolic names for the DEC Standard 144 device model numbers. These symbolic names take the form of *DT$_xxx*, where **xxx** is the device name.

The complete table of DEC Standard 144 disk device model numbers known to the VMS disk class driver is in the routine titled Media-id to Device Type Conversion Table in VMS module DUDRIVER. Table 5–4 summarizes those that relate to disk devices.

Table 5–4: DEC Standard 144 Disk Device Codes

MODEL	Value (decimal)	MODEL	Value (decimal)
DT$_RKO6	1	DT$_RZ24	50
DT$_RK07	2	DT$_RZ55	51
DT$_RP04	3	DT$_RRD40S	52
DT$_RP05	4	DT$_RRD40	53
DT$_RP06	5	DT$_RX23	55
DT$_RM03	6	DT$_RF31	56
DT$_RP07	7	DT$_RF72	57
DT$_RP07HT	8	DT$_RAM_DISK	58
DT$_RL01	9	DT$_RZ25	59
DT$_RL02	10	DT$_RZ56	60
DT$_RX02	11	DT$_RZ57	61
DT$_RX04	12	DT$_RX23S	62
DT$_RM80	13	DT$_RX33S	63
DT$_RM05	15	DT$_RA92	64

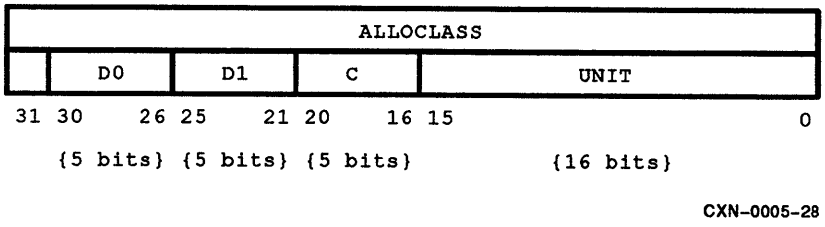
Table 5–4 (Cont.): DEC Standard 144 Disk Device Codes

MODEL	Value (decimal)	MODEL	Value (decimal)
DT$_RX01	16	DT$_RZ23L	66
DT$_RB02	18	DT$_RX26	67
DT$_RB80	19	DT$_RZ57I	68
DT$_RA80	20	DT$_RZ31	69
DT$_RA81	21	DT$_RZ58	70
DT$_RA60	22	DT$_SCSI_MO	71
DT$_RC25	23	DT$_RWZ01	71
DT$_RCF25	24	DT$_RRD42	72
DT$_RD51	25	DT$_CD_LOADER_1	73
DT$_RX50	26	DT$_ESE25	74
DT$_RD52	27	DT$_RFH31	75
DT$_RD53	28	DT$_RFH72	76
DT$_RD26	29	DT$_RF73	77
DT$_RA82	30	DT$_RFH73	78
DT$_RD31	31	DT$_RA72	79
DT$_RD54	32	DT$_RA71	80
DT$_RRD50	34	DT$_RAH72	80
DT$_RX33	36	DT$_RF32	81
DT$_RX18	37	DT$_RF35	81
DT$_RA70	38	DT$_RFH32	82
DT$_RA90	39	DT$_RFH35	82
DT$_RD32	40	DT$_RFF31	83
DT$_R009	41	DT$_RF31F	83
DT$_RX35	42	DT$_RZ72	84
DT$_RF30	43	DT$_RZ73	85
DT$_RF70	44	DT$_RZ35	86
DT$_RF71	44	DT$_RZ24L	87
DT$_RD33	45	DT$_RZ25L	88
DT$_ESE20	46	DT$_RZ55L	89
DT$_RZ22	48	DT$_RZ56L	90
DT$_RZ23	49	DT$_RZ57L	91

In order to ensure that the Disk Unit Identifiers are truly unique in a VAXcluster environment, their format as supplied by the VMS based MSCP server incorporates the allocation class of the device being served as well as the entire unit name. This format applies only to unit identifiers supplied by a VMS based MSCP server and not to an MSCP speaking controller such as an HSC or KDM.

Figure 5–8 shows the format of a unit identifier supplied by a VMS based MSCP server:

Figure 5–8: Unit Identifier Format for VMS Based MSCP Servers



Field	Description
ALLOCLASS	The allocation class for the VAX in which the VMS–based MSCP server resides
D0	The first letter of the device name: **D**dcu
D1	The second letter of the device name: d**D**cu
C	The controller letter in the device name: dd**C**u
UNIT	Actual drive unit number used to form the device name: ddc**U**

The letters from the device name represented by D0 , D1 , and C are normalized to be numbers in the range of 1 through 26 *(RAD Hustvedt)*. Thus, "A" is represented by a "1", "B" is represented by a "2", ..., "S" is represented by a "19" (decimal), etc.

Bit 31 in this unit identifier format is currently undefined for VMS V5.5 and is reserved for future use.

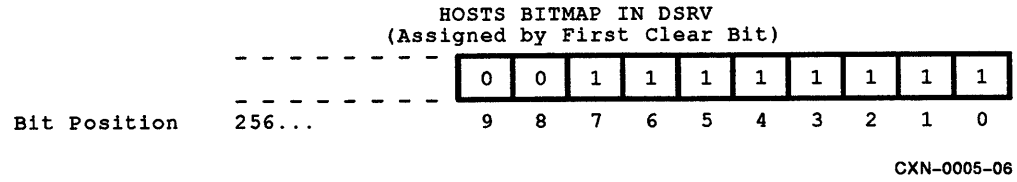
5.4.3 Host Numbers

When the local MSCP server establishes an SCS connection with a remote disk class driver, it assigns a *Host number* to that remote host. This is done by searching a bitmap at offset *DSRV$B_HOSTS* in the DSRV, starting with bit 0 for a maximum of *256 bits* for the first free bit.

The bit number of the first clear bit is assigned to the remote host as its host number; this number is stored at offset *HQB$B_HOSTNO* in the HQB for the remote host. To prevent the same host number from being assigned to another host, the bit in the bitmap is enabled and set to a "1".

Figure 5–9 illustrates that "8" is the host number to be assigned to the next remote host with which an SCS connection is established.

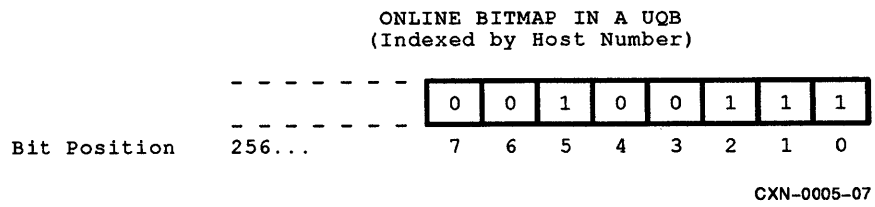
Figure 5–9: Host Index Bitfield at DSRV$B_HOSTS in the DSRV



At offset *UQB$B_ONLINE* in each Unit Queue Block is a bitmap representing which remote hosts have the unit online. Part of processing of an *ONLINE* command from a remote host is to fetch its host number from the HQB$B_HOSTNO field in its HQB, use this host number as an index into the UQB's ONLINE bitmap, and then set the bit corresponding to the host number. If a unit leaves the online state relative to the remote host, then that same bit is cleared.

Figure 5–10 illustrates that the unit is online to remote hosts 0, 1, 2, and 5, but not online to remote hosts 3, 4, 6, and 7.

Figure 5–10: UQB's Online Field Bitmap of Hosts Accessing a Specific Unit



5.4.4 Transfer Buffers

From the local MSCP server's point of view, writing data to a disk that is being served on behalf of a remote host can be thought of as a two phase operation:

- First, the data must be received from the remote host into a buffer in the local host's memory.
- Second, the data is actually transferred to the disk by building an IRP which references the buffer, and queuing the IRP to the driver for the disk.

Similarly, the MSCP server perceives reading data from a disk being served on behalf of a remote host as being another two phase operation:

- The data is first transferred into a buffer residing in the local host's memory.
- Then it is transferred from the buffer to the remote host.

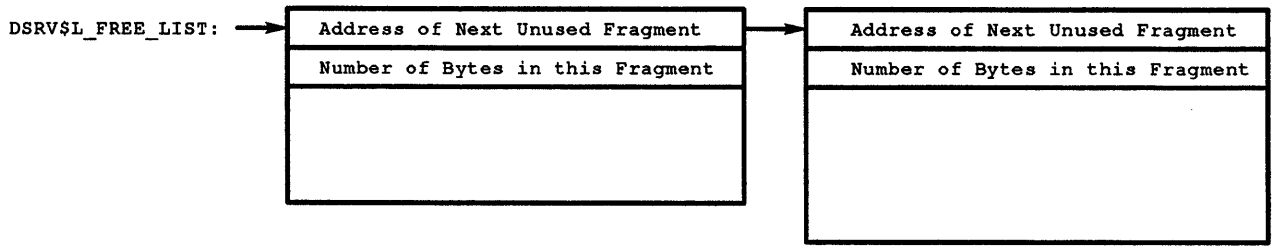
Buffers used by the MSCP server as temporary holding areas for data being transferred between a local disk and a remote host are called *Transfer Buffers.*

5.4.4.1 Transfer Buffer Allocation

When the MSCP server is first loaded and started, it allocates its own pool of transfer buffers from general nonpaged pool. Then, "fragments" of this pool are dynamically allocated and released on an "as needed" basis. Unused fragments in this private pool are maintained in a linked list whose head is at offset *DSRV$L_FREE_LIST* in the DSRV.

The first longword of each fragment contains the address of the next fragment in the list, with the last fragment in the list having a zero in this field. The second longword contains the number of bytes in the fragment itself. Figure 5–11 illustrates an unused fragment link:

Figure 5–11: Free Transfer Buffer Linkage



When the the buffer pool is first allocated, it is set up as a standard VMS data structure whose bytes are all contained within one fragment. Figure 5–12 displays the initial buffer contents.

Figure 5–12: Initial State of the Transfer Buffer

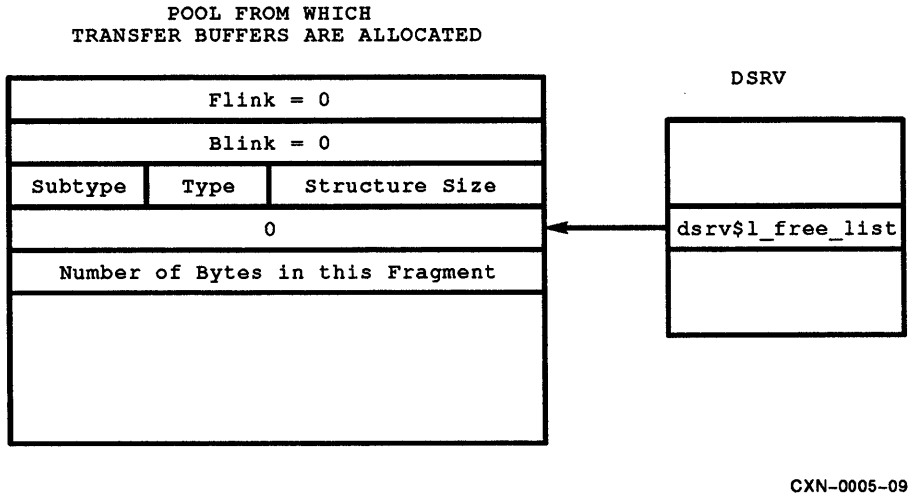
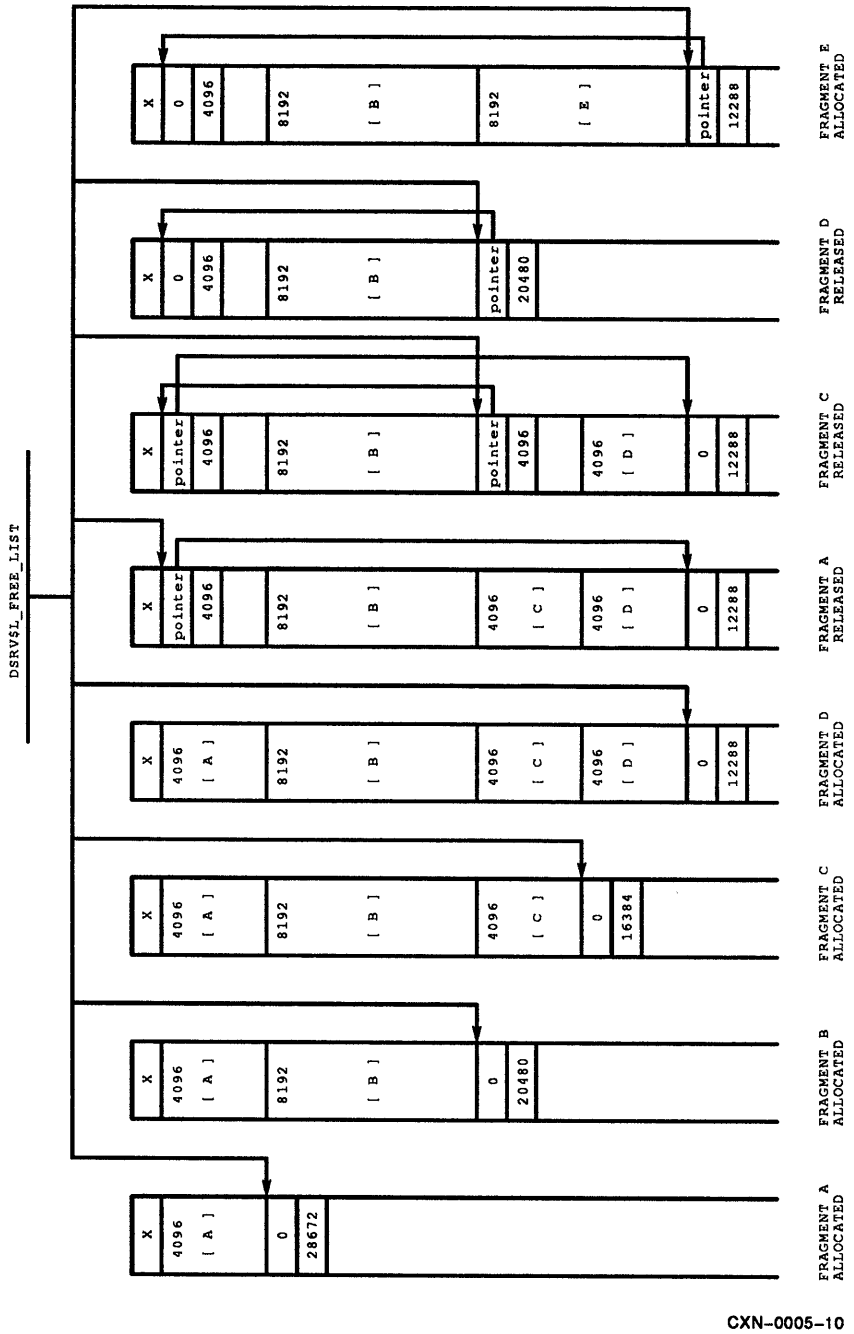


Figure 5–13 shows what happens as fragments are allocated from, and released back to the pool. The diagram assumes that the pool initially contains 32768 free bytes.

- The first four columns illustrate the allocation of fragment A (4096 bytes), fragment B (8192 bytes), fragment C (4096 bytes), and fragment D (4096 bytes).
- The next two columns show first, the release of fragment A, and then fragment C, and that they have been placed back in the list of free fragments.
- In the seventh column, fragment D has been released. But it was also merged with other free fragments with which it was contiguous. This, in turn, caused fragment A to become the end of the list.
- Finally, the eighth column illustrates that an 8192-byte fragment E is allocated by essentially "deducting" it from the first free fragment that is at least as large as what is being allocated.

Figure 5–13 illustrates this senario:

Figure 5–13: Transfer Buffer Allocation and Deallocation



The size of the local pool of transfer buffers allocated by the VMS based MSCP server is governed by the *Sysgen Parameter MSCP_BUFFER*

When the VMS based MSCP server is started, the number of pages specified by the MSCP_ BUFFER parameter are allocated from nonpaged pool and reserved for buffer usage. The value of this parameter is kept at System Location *CLU$GL_MSCP_BUFFER*

Routine *ALLOCATE* is called to allocate a transfer buffer from this pool. It searches the the list of free fragments looking for the first one which is at least as large as what is being requested. If insufficient pool exists, the following process is followed to allocate a buffer.

The VMS based MSCP server is capable of dynamically calculating an upper and lower limit to the size of a given transfer buffer that is to be allocated for a particular transfer. It uses the following procedure to compute this size:

- The server computes the number of bytes it will initially attempt to allocate for the transfer buffer based on the following:
  - — The smallest multiple of 512 bytes greater than or equal to the total transfer size is computed. This is done by computing the sum of the total transfer size plus 511, and then clearing the low order 9 bits of this sum.
  - — The larger of two numbers is computed: one half the number of unused bytes in the server's local pool, or 512 bytes.
  - — The initial number of bytes the server will attempt to allocate, is then computed as the smaller of the first value calculated (actual buffer size required) and the second value calculated (the greater of one half the number of unused bytes or 512)

  If the total transfer size is greater than the available pool, then the server will break the request into multiple transfers.
- Next, code is entered to actually allocate a transfer buffer consisting of the required number of bytes from the server's local pool.

  This code scans the linked list of unused "chunks" in the pool, looking for the first chunk containing at least the required number of bytes.
  - — If the chunk contains exactly the number of bytes, then it is removed from the list and used as the transfer buffer.
  - — If the chunk contains more than the required bytes, then the first required bytes of the chunk are used for the transfer buffer. The remainder is made into a smaller chunk and left in the list of unused chunks.

  If there are no chunks containing at least the required bytes, then this value is reduced to half its original value, and the code to allocate a transfer buffer from the list of unused chunks is again entered.

  To prevent the server's local pool from being heavily fragmented, the value of the required bytes is tested the first and each successive time this code is entered. If the value satisfies both of two conditions, then the I/O request is suspended. These two conditions are as follows:
  - — Required size is less than 1/4 the total size of the transfer.
  - — Required size is less than the content of the *DSRV$L_BUFFER_MIN* field in the server's DSRV.

  The deallocation of some other transfer buffer back to the pool triggers this code to attempt once again the allocation of a transfer buffer to the request.

During MSCP server initialization, the DSRV$L_BUFFER_MIN field is set to contain one eighth the total size of the pool.

5.4.5 VMS based MSCP server Flow Control

The Server uses the same credit scheme for *Flow Control* as does a remote MSCP server. An SCS buffer containing a received MSCP command is not released back to the port by the server until the server is finished with the command. The Sysgen Parameter *MSCP_CREDIT* specifies the number of commands that a server can have active from any individual disk class driver. The default value for VMS V5.5 is set at eight. This value is stored at System location *CLU$GL_MSCP_CREDITS*

5.4.6 Controller Timeout

The VMS based MSCP servers *Controller Timeout Interval* is hard-coded in module MSCP.MAR, routine SET_CONTROLLER_CHAR to be 20 seconds.

5.4.7 MSCP Server Initialization Overview

There are three major aspects of server initialization:

- Loading and Starting the MSCP Server.

  This accomplishes four major tasks.

  — The pool of transfer buffers is allocated.

  — The DSRV is initialized.

  — The server declares to the SCS layer that it is ready to establish SCS connections with remote disk class drivers (SCS Listen Routine)
- Start the *Load Monitoring Thread* if appropriate
- Setting Disks Served to Remote Hosts.

  After the MSCP server has been loaded and started on the local host, as new devices are discovered by the *Configure Process*, the System location *CLU$GL_MSCP_SERVE_ALL* is examined to determine if *Auto-Serving* is turned on. The value of this location is determined by the Sysgen Parameter *MSCP_SERVE_ALL*

  The MSCP_SERVE_ALL parameter may have one of the following values:

Value	Description
0	Do not serve any disks. This is the default.
1	Serve all available disks.
2	Serve only locally-attached (non-HSC) disks.

Optionally, the DCL command SET DEVICE/SERVED is used to enter a local disk unit into the database of served disks.

When a device is discovered, a UQB is allocated and initialized for that unit. The UQB is inserted into the queue of UQBs attached to the DSRV in unit number order; the lower the unit number, the closer the UQB is to the head of the queue. Finally, an AVAILABLE ATTENTION message is sent to each remote disk class driver with which the local MSCP server has an SCS connection. The "database of served disks" is effectively the queue of UQBs attached to the DSRV.

- Accepting an SCS CONNECT Request from a Remote Host.

  When a remote disk class driver wishes to establish an SCS connection with the local MSCP server, the server allocates and initializes an HQB corresponding to the remote host, ACCEPTs the CONNECT request, and then inserts the HQB at the end of the DSRV's queue of HQBs.

The next three sections present the major details of each of these three aspects of MSCP server initialization.

5.4.8 Loading and Starting the MSCP Server

At system startup time, the STACONFIG process examines the value of the System location *CLU$GL_MSCP_LOAD* (Sysgen Parameter *MSCP_LOAD*). If a nonzero value is found, the VMS based MSCP server software will be initialized and started at routine *START*. The following is performed for the server's initialization.

- The DSRV structure is initialized
- The minimum transfer buffer size constant is initialized to be 1/8 of the total buffer pool and is stored in the *DSRV$L_BUFFER_MIN* offset of the DSRV.
- The server's pool of transfer buffers are allocated
  - — The size of the pool is stored in the *SRVBUF$L_SIZE*. offset of the buffer pool.
  - — The address of the pool is stored at the *DSRV$L_SRVBUF* offset in the DSRV.
  - — The sum of the Free Buffer Fragments is stored in the DSRV at offset *DSRV$L_AVAIL*
  - — The pool is initialized as one large free fragment; and the pool's free fragment listhead, DSRV$L_FREE_LIST, is initialized to contain the address of that fragment.

NOTE

Failure to allocate the nonpaged pool required for this step results in an SS$_INSFMEM (insufficient memory) error, and server initialization is terminated. This will be distinguished by clearing the address of the DSRV stored in location SCS$GL_MSCP.

- The HQB, UQB, and MEMW (memory wait) queues in the DSRV are initialized as being empty.
- The Load Balancing fields are initialized
- The HULB queue in the DSRV is initialized as being empty
- The multi-host controller flag, CF_MLTHS and the Load Balance flag, CF_LOAD are set in the DSRV$W_CFLAGS field of the DSRV.

- A Unique Identifier is constructed for the controller based on the hosts SCSSYSTEMID, the model type of "Emulator" and class type of "MSCP Controller" and is placed in the CTRL_ID offset of the DSRV.
- The MSCP server invokes the SCS LISTEN service to place itself in the SCS list of "listening SYSAPs". In essence, it is declaring itself ready and willing to converse with other SYSAPs, specifically remote disk class drivers, via SCS connections.
- The Load Monitoring Thread is started if we are serving non-local disks.

5.4.9 Serving Devices

Both the DCL command "Set Device/Served" and the Configure Process call routine *MSCP$ADDUNIT* when a servable disk unit is made known.

Routine MSCP$ADDUNIT allocates and initializes the UQB, and inserts it into the DSRV's queue of UQBs. It then calls routine *ADD* in the MSCP server to actually notify remote hosts that the unit just set served is now available to them.

- Routine MSCP$ADDUNIT begins by verifying that the MSCP server has been loaded. (If it hasn't, it merely returns the error SS$_DEVOFFLINE and does no further processing.)
- Validations are made to verify that the disk is a valid candidate for serving:
  - — The device is cluster wide device (noclu flag in the UCB clear)
  - — Device is not already served
  - — The device is of the class *DC$_DISK*.
  - — System allocation class matches that of device
  - — If allocation class of the device is zero only serve if local
  - — Check that the device is not mounted cluster accessible
- The queue of UQBs attached to the DSRV is searched for a UQB corresponding to the unit to be set served.
  - — If one is found, then MSCP$ADDUNIT merely returns the VMS condition value SS$_NORMAL and does no further processing. The unit has already been set served.
  - — If a matching UQB is not found, then then one is allocated from nonpaged pool.

NOTE

If nonpaged pool is not available, MSCP$ADDUNIT returns the "insufficient memory" condition value SS$_INSFMEM.

- The new UQB is now initialized. In particular:
  - — The STATE field is set to AVAILABLE.
  - — The UNIT number field is initialized.
  - — The Server Local Unit Number is assigned
  - — Set the SLUN bit in Unit Number indicating a Server Local Unit.
  - — Set the DEVNAME field of the UQB
  - — Force bad block replacement flag for all served devices
  - — The software and/or hardware write protect flags, UF_WRTPS and UF_WRTPH, are set in the FLAGS field if appropriate.

— The queue of blocked HRBs, waiting for sequential commands to complete, is initialized as being empty.

— A unique device name is created using routine *IOC$CVT_DEVNAM*

- The UQB is inserted into the queue of UQBs attached to DSRV.

NOTE

This queue is maintained in unit number order. The lower the unit number of the UQB, the closer the UQB is to the head of the queue.

- Calls routine ADD in the MSCP server to send an AVAILABLE ATTENTION message to each remote host for which there is an HQB queued to the DSRV.

5.4.10 ACCEPTing an SCS CONNECT From a Remote Host

When the local MSCP server completed its initialization, it used the SCS service LISTEN to place itself in the SCS list of "listening SYSAPs". One of the arguments it passed to the LISTEN service was the address of its routine to which the SCS layer should pass incoming CONNECT requests. The name of this routine is *LISTN*.

Routine LISTN is passed the packet containing the CONNECT request; so it knows from where the request originated. The major steps taken by MSCP server routine LISTEN are as follows:

- It allocates an HQB to represent the host whose disk class driver sent the CONNECT request.

  This allocation is from nonpaged pool. If it fails due to a lack of sufficient nonpaged pool, then the CONNECT request is REJECTed and no further processing will be done for the request. If the allocation succeeds, then routine LISTN continues servicing the request in the following manner:

  — Most of the fields in the HQB are cleared to 0.

  — The HQB's queue of HRBs is initialized as being empty.

  — The first clear bit in the DSRV's HOSTS bitmap is found. The position of this bit relative to the beginning of the bitmap will serve as the "host number" of the remote host which sent the CONNECT request.

    o The bit is set in the HOSTS bitmap so that the same host number will not be given out to some other remote host.

    o The bit position number is stored in the HQB$B_NOSTNO field. The bit position number is now the remote host's "host number".

    o The default host timeout value, 60 seconds for VMS V5.5, is stored in the HQB$W_HTIMO field.

NOTE

If all the bits in the HOSTS bitmap are already in use, then the CONNECT request is rejected, the HQB is deallocated, and no further processing is performed by this routine. The Hosts bitmap consists of 32 bytes providing service for up to 256 hosts in VMS V5.5.

— The HULB vector is allocated and initialized to zeros

- Routine LISTN now invokes the SCS service ACCEPT to actually accept the CONNECT request.
  — One of the arguments passed to the ACCEPT service is the address of the MSCP server's routine (MSG_IN) for handling all incoming SCS messages on this connection. These messages will be the MSCP commands sent to the local server by the remote disk class driver at the "other end" of the connection.
    o The SCS layer will allocate and initialize a CDT (Connection Descriptor Table) to describe the connection, and store in the CDT the address of the server's message input routine.

      When an MSCP command arrives via this connection, the SCS layer is able to identify the routine to which the command is to be passed; it is the message input routine whose address is in the CDT associated with the connection.
    o The local MSCP server maintains one connection for each remote disk class driver with which it converses in MSCP. However, routine MSG_IN is the common message input routine for all such connections. This is permissible since each incoming message contains the identities of the host from which the message originated and the SYSAP (i.e. disk class driver) in that host which sent the message.
  — Another of the parameters passed by the MSCP server to the ACCEPT service is the number of send credits to extend to the remote disk class driver. This is initialized to contain the value stored at System location CLU$GL_MSCP_CREDITS.

NOTE

If the ACCEPT fails for any reason (e.g. SCS protocol failure), then the HQB will be deallocated, the bit position in the HOSTS bitmap will be cleared, a REJECT of the request will be performed and no further processing will be done for the CONNECT request.

- The HQB is inserted at the end of the queue of HQBs maintained by the DSRV.
- The address of the CDT describing the connection with the remote disk class driver is stored in the HQB.

5.5 MSCP Server Load Balancing

5.5.1 Static Load Balancing

VMS V5.4 introduced the concept of VMS based MSCP server *Static Load Balancing*. The initial implementation is designed to distribute the VMS based MSCP server's work load across multiple hosts that can provide access to a given device. Load Balancing is performed on the level of each unique host and unit combination during the initial mount of the device from a host, or during mount verification.

In a Mixed Interconnect VAXcluster, there will be situations where some hosts will have Direct Access to a device that is being served from a multihost controller such as an HSC or ISE and others will not. Other hosts within the VAXcluster that do not have a direct path will rely on the hosts with the direct path to act as Disk Servers to provide access to the devices. Disk Serving is provided by the VMS based MSCP server.

For any given disk on a multihost controller, there may be more than one host node with a direct path. Provided multiple hosts with a direct path are running the VMS based MSCP server and are offering service for the disks to the remaining VAXcluster members, there may be several viable paths to the disks through the Disk Servers for the members without a direct path.

Static load balancing was introduced to attempt to equitably distribute the work load performed by the VMS based MSCP servers on behalf of the VAXcluster members without a direct path to a given disk.

Each VAXcluster member acting as a disk server for a device on a multihost controller defines its ability to perform the work of serving in terms of its *Load Capacity*. The load capacity of a given host is the number of Read and Write operations per second that this host should be capable of performing for its served devices.

The Sysgen parameter MSCP_LOAD may be used to specify the load capacity for a host by representing it as a positive value greater than one. If a load capacity is not explicitly set for a host, a default value will be assigned to it in routine *LM_INIT_CAPACITY* as part of the VMS based MSCP server's initialization. The Load Capacity is stored in the DSRV at offset *DSRV$W_LOAD_CAPACITY*. Table 5–5[1] lists the default load capacity values assigned for the current VAX family of processors.

[1] Table values reflect VMS V5.5-2 entries

Table 5–5: Default Load Capacity

Load Capacity	VAX Processor
20	MicroVAX 2000, VAXstation 2000
45	VAX 11/750,VAXft 3000, MicroVAX 3100, VAXstation 3100, VAXstation 3500
60	VAX 82xx, VAX 83xx
70	VAX 11/780
80	MicroVAX II, VAXstation II
100	VAX 11/785, VAX 86xx
120	MicroVAX 3500, MicroVAX 3600, MicroVAX 3800, MicroVAX 3900
130	MicroVAX 3300, MicroVAX 3400
200	VAX 6000-200, VAX 6000-300
325	VAX 4000-300
340	VAX 85xx, VAX 8700, VAX 88xx
400	VAX 4000-400, VAX 4000-600, VAX 6000-400, VAX 6000-500, VAX 6000-600, VAX 7000, VAX 9000-xxx
100	All Other Systems

When a Disk Class Driver on a host without a direct path to a device attempts to place a given unit online to that host either by a Mount request or through Mount Verification, the Disk Serving host which will be impacted the least is chosen as the Disk Server. The decision as to which Disk Server to use for this particular host and unit combination is based on each VMS based MSCP server's *Load Availability*.

Load Availability is defined as the host's Load Capacity minus the current total Operation Count being performed for all host and unit combinations on a given host. The Operation Count is derived by taking the average Read and Write request rate for each host and unit combination over the past 20 second interval. The host with the highest Load Availability will be chosen as the Disk Server for the new host and unit combination.

The selection of the appropriate Disk Server is only performed at the time of a mount request or during mount verification, so this type of load balancing is known as *Static Load Balancing*.

Dynamic Load Balancing will be defined as the ability of each host and unit combination to be switched to other Disk Servers based on the current load being placed on the VMS based MSCP Server. This form of load balancing is planned for a future release of the VMS operating system and is not present in VMS V5.5.

5.5.2 Load Monitoring Thread

The *Load Monitoring Thread* (Routine LOAD_MONITOR) is responsible for calculating the current load available for the VMS based MSCP server. This is performed by walking the HULB structures every *Load Monitor Interval* seconds and totaling the operation count for each host and unit combination. The Load Monitor Interval is currently hardcoded in routine *LM_INIT* as twenty seconds for VMS V5.5.

The Load Monitor Thread is executed as a fork thread. It is scheduled to execute by building a repeating Timer Queue Entry (TQE) in the *Load Monitor Initialization Routine* (LM_INIT). The timer queue entry is scheduled to execute every twenty seconds (VMS V5.5).

When the timer queue entry is signaled, routine *MSCP$TMR* is called. Routine MSCP$TMR creates the LOAD_MONITOR fork thread.

The Load Monitor fork thread walks the Host Unit Load Block (Hulb) queue that is linked from the DSRV at offset DSRV$L_HULB_FL. For each valid Hulb encountered, the Operation Count field (*HULB$W_OPCOUNT*) is added to a total operation count. The current contents of the Opcount field are stored in the *HULB$W_PREV_OPC* field of the Hulb and the Opcount field is set to zero.

When the end of the Hulb queue is reached, the previous three intervals statistics are shifted in the DSRV$W_LM_LOAD fields of the DSRV. When the operation count is finally calculated for the load availability, it will actually be the average of the current plus the last three intervals average operation counts.

The total Operation Count for this interval is divided by the elapsed time period to provide the average operations for this load balance interval. This value is stored in the *DSRV$W_LM_LOAD1* field of the DSRV. The operation count for this interval is then calculated as the average of the previous three intervals (DSRV$W_LM_LOAD2 through DSRV$W_LM_LOAD4) plus this intervals operation count.

The Load Availability for this VMS based MSCP server is than calculated as the server's Load Capacity minus the current Operations Count. This result is stored in the DSRV at offset *DSRV$W_LOAD_AVAIL*.

For a future release of VMS, this is where a check will be made to determine if Dynamic Load Balancing is required. If the Load Availability is less than the *Load Threshold* (currently defined as ten I/Os per second) the Load balance thread (LOAD_BALANCE) will be executed to perform dynamic load balancing. This code is currently not implemented.

The Load Monitoring thread than exits and will be re-executed during the next load monitor interval.

5.6 MSCP Server's Handling of READ and WRITE Commands

The purpose of the VMS based MSCP server is to make disks that are directly accessible to one host available for reading and writing to nodes that do not have direct access to the device. The next few sections of this chapter discuss how the local server handles MSCP READ and WRITE commands from remote disk class drivers.

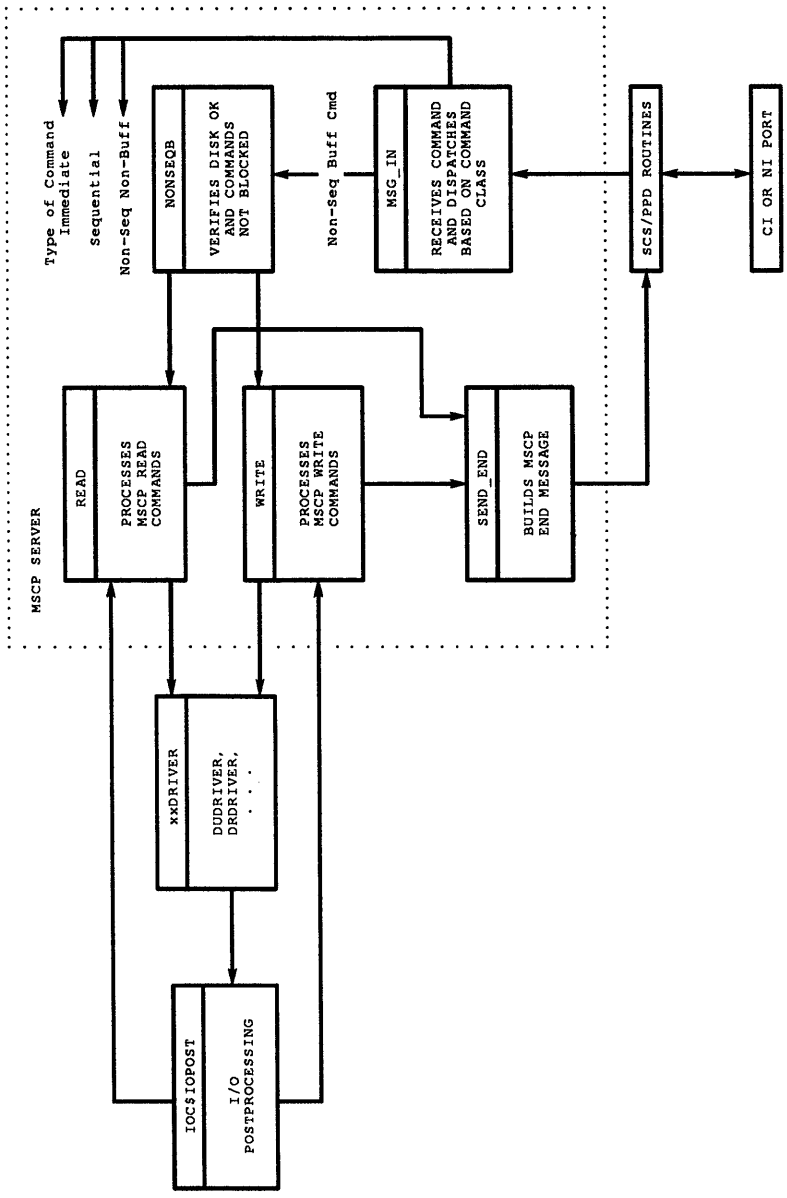
The VMS based MSCP server is required to perform the interim handling of data on behalf of the remote node. In processing a READ command, the MSCP server allocates a local buffer into which the data can be read from the disk. The server must then pass the data on to the host which requested it.

Similarly, in processing a WRITE command, the server must again allocate a temporary local buffer and request the data from the remote host. Once the data arrives, it is then transferred from the buffer to the requested disk. This involves interactions between the MSCP server and the SCS layers for communication with the remote disk class driver, and interactions between the server and the local driver for the particular disk unit involved.

In the next few sections, overviews are presented of the server's handling of READ and WRITE commands. Then, details of the routines involved in each are provided.

Figure 5–14 illustrates the general flow of both a READ and a WRITE command. It should be referred to when reading the overviews and detailed descriptions of both of these operations.

Figure 5–14: General Flow of VMS based MSCP server Reads and Writes



CXN-0005-11

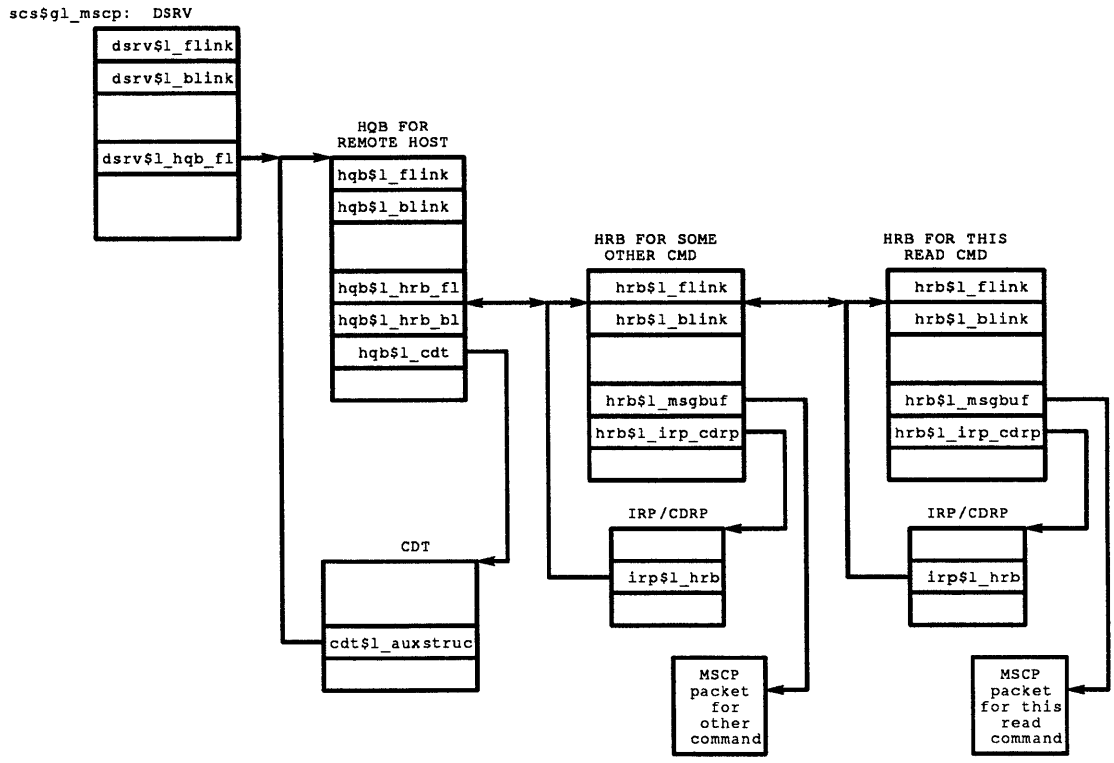
5.6.1 Overview of MSCP Server Handling READ Command

The MSCP server's handling of a READ command from a remote host begins when the SCS layer passes to the server's message input routine, *MSG_IN*, the MSCP packet containing the READ command. MSG_IN is also handed the address of the CDT representing the connection with the host which sent the command, and the address of the PDT describing the port through which the command was received.

MSG_IN allocates an HRB and IRP from nonpaged pool to represent the READ command, and stores the address of the IRP in the HRB. The address of the HQB corresponding to the host which sent the READ command is copied into the HRB from the *Auxiliary Structure* field of the CDT representing the SCS connection with the remote host. The address of the MSCP READ command packet itself is also stored in the HRB. The opcode, length, modifiers, and flags in the READ command are validated, and then the HRB is inserted into the HQB's queue of HRBs.

Figure 5–15 illustrates the data structures and the linkages that are involved in this step.

Figure 5–15: Data Structures and Linkage Involved in a Server Receiving a Command



MSG_IN now dispatches to one of five routines based on the *Class Field* within the opcode of the MSCP command:

Routine	Class of Commands Handled by Routine
IMMEDIATE	Immediate Class Commands
SEQUENTIAL	Sequential Class Commands
NONSEQ	Nonsequential Nonbuffered Class Commands
BAD_OPC	Maintenance Commands (none supported by this server)
NONSEQB	Nonsequential Buffered Class Commands

A READ command is a *Nonsequential Buffered Command*. Nonsequential commands are commands which "MSCP speaking" controllers can re-order for purposes of performance optimization. They may also be segmented, and their segments interleaved with other nonsequential commands.

A nonsequential command is considered "buffered" if it requires the controller to allocate one or more intermediate buffers to temporarily hold the data which is being exchanged between a unit and a remote host. (A WRITE command is another example of a nonsequential buffered MSCP command.)

The Message Input routine, *MSG_IN*, dispatches to routine *NONSEQB*, passing it the address of the MSCP command and the HRB corresponding to that command.

NOTE

Maintenance Commands are not supported by the VMS based MSCP server. If one is received, MSG_IN branches to routine *BAD_OPC* to issue an End Message with the status code *MSCP$K_ST_ICMD* (invalid command).

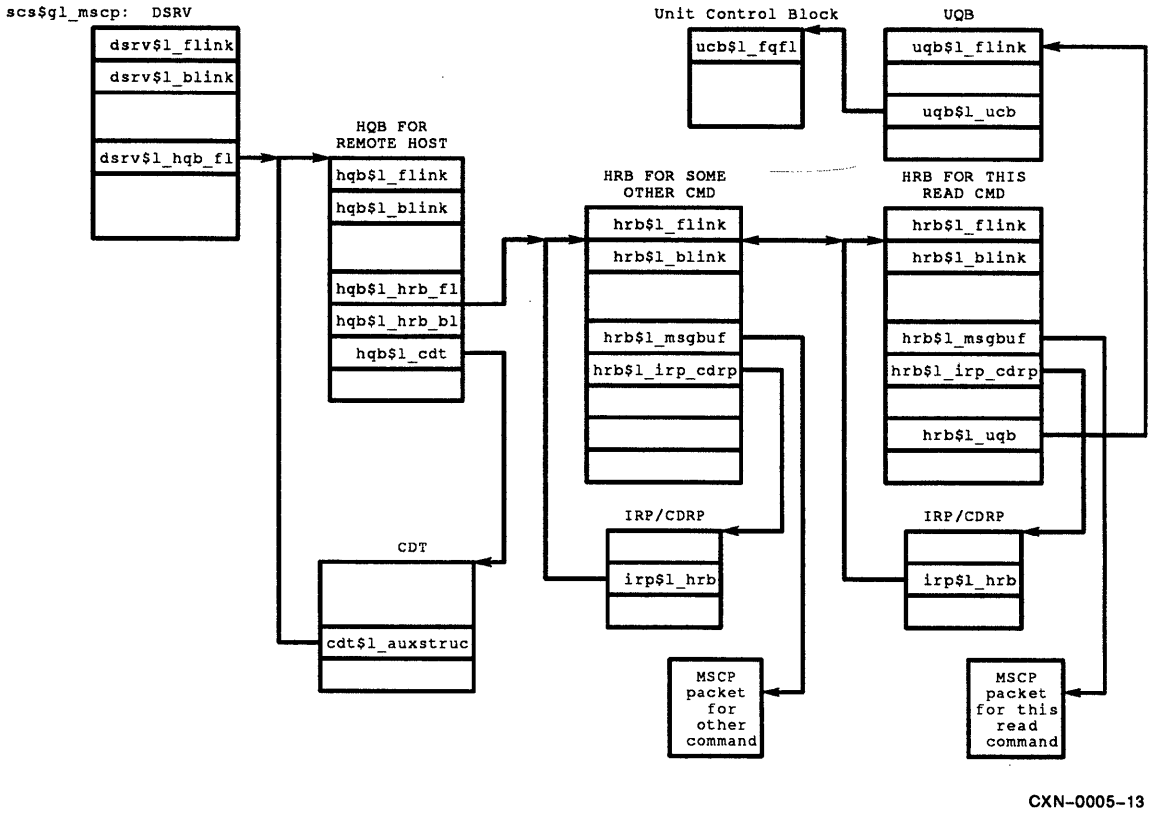
The three remaining classes of commands, Immediate, Sequential, and Nonsequential Nonbuffered, are covered later in this chapter.

Routine NONSEQB scans the queue of UQBs representing units served by this MSCP server, looking for one whose unit number matches the one in the READ command. The address of the UQB is stored in the HRB. NONSEQB verifies that the unit is online. It then dispatches to routine *READ* to process the READ command. (NONSEQB would have dispatched to routine *WRITE*, had this been a WRITE command.)

Passed to routine READ by routine NONSEQB are the addresses of the MSCP command, the HRB representing the command, and the UQB corresponding to the unit which is referenced by the command.

Figure 5–16 illustrates the linkages among the MSCP server data structures with which routine READ works. (This same illustration would also apply to a WRITE command as well, provided, of course, that the word "READ" is changed to "WRITE".)

Figure 5–16: Data Structures Involved with MSCP Read and Write Commands



Routine READ copies the total size of the disk transfer to the *Original Byte Count* field of the HRB. It then allocates a transfer buffer from the MSCP server's local nonpaged buffer pool. This buffer will be used to temporarily buffer the data being read from the disk until it is transferred to the host which issued the READ command.

The content of this buffer will be transmitted to the remote host by means of a local CI, DSSI or NI port. It will be mapped using mapping resources specific to PADRIVER, PIDRIVER or PEDRIVER. The local buffer handle resulting from this mapping is stored in the HRB at offset *HRB$B_LBUFF* .

Next, routine READ prepares the IRP for at least the first segment of the disk transfer (or the entire transfer if only one segment is involved), initializing the UCB, SVAPTE, BOFF, and BCNT fields of the IRP. The IRP's FUNC field is set to *IO$_READPBLK*.

Routine READ then calls *DO_DISK*. DO_DISK stores the address of a special I/O postprocessing routine, *BACK*, in the IRP's PID field. It also stores the address of the instruction following the call to DO_DISK in the HRB at offset *HRB$L_RESPC*. It then stores the starting LBN in the IRP, and queues the IRP to the driver for the unit by jumping to routine *EXE$INSIOQC*.

The driver handles the IRP just as if the request had originated on the local node. The data is read from the disk into the transfer buffer mapped above, and eventually I/O postprocessing is invoked. IOC$IOPOST passes the IRP to routine BACK, which in turn resumes the request within the MSCP server at the instruction in routine READ following the call to DO_DISK.

Routine READ then increments the operation count in the Hulb structure. It then initializes the CDRP attached to the IRP to transmit to the remote host the data just read in from the disk. The CDRP is set up such that:

- The number of bytes to transfer to the remote host is the number of bytes just read from the disk.
- The remote host's buffer handle is in the original MSCP read command. So the address of this buffer handle is copied into the CDRP.
- The offset into the remote node's buffer is merely the accumulated byte count read from the disk thusfar.

NOTE

This field is kept in the HRB, and was initialized to 0 when the HRB was allocated.

- The local buffer handle in the CDRP is the local buffer handle in the HRB used to map the transfer buffer into which the data was read.

SCS routines are then called to allocate an RSPID, allocate an SCS message buffer, build a send data (*SNDDAT*) command in the message buffer, and queue the SNDDAT to the local port. These SCS routines are invoked by the *SEND_DATA macro* and are provided the CDRP as a basis for doing their work.

The SNDDAT command causes the local port to transmit the data in the transfer buffer to the remote host. The request is suspended as a fork thread until the port completes the transmission. When it resumes, it adds into the accumulated byte count field in the HRB the number of bytes just sent to the host.

If the entire request is not complete, then routine READ branches back to prepare another IRP for the next segment of the request. If the request is complete, then an MSCP End Message bearing a success status code is sent to the remote host, all resources held by the HRB are released, and the HRB and IRP are deallocated.

5.6.2 Overview of MSCP Server Handling WRITE Command

The MSCP server's handling of a WRITE command from a remote host begins when the SCS layer passes to the server's message input routine, MSG_IN, the WRITE command. Routines MSG_IN and *NONSEQB* process a WRITE command exactly as they would a READ command, except that NONSEQB dispatches it to routine *WRITE* instead of READ.

Routine WRITE copies the total size of the transfer into the Original Byte Count field of the HRB. It then allocates a transfer buffer from the MSCP server's local nonpaged buffer pool. This buffer will be used to temporarily store the data being sent by the remote host until it is transferred to the disk for which the WRITE command was issued.

Since the data is received by means of a local CI, DSSI or NI port, the transfer buffer is mapped using mapping resources specific to PADRIVER, PIDRIVER or PEDRIVER. The buffer handle produced by this mapping is stored in the HRB.

Routine WRITE then increments the operation count in the Hulb structure. It then initializes the CDRP attached to the IRP so that it may be used to request data from the remote host. This is done in the same manner as when the READ command initialized the CDRP to transmit data to the remote host.

SCS routines are called to allocate a RSPID, allocate an SCS message buffer, build a REQDAT (request data) command in the message buffer, and queue the REQDAT to the local port. These SCS routines are invoked by the *REQUEST_DATA macro* and base the content of the REQDAT on the information contained in the CDRP.

The request is suspended as a fork thread until the data is received in the transfer buffer by the local port from the remote host. When the request resumes, routine WRITE prepares the IRP for the first segment of the disk write request (or possibly the entire request if all the data was requested from the remote host at once). This involves setting up the UCB, SVAPTE, BOFF, and BCNT fields in the IRP, and setting the IRP's FUNC field to "WRITE".

Routine WRITE calls DO_DISK to actually actually execute the disk transfer. DO_DISK stores the address of the special I/O postprocessing routine BACK in the IRP's PID field. It also stores the address of the instruction following the call to DO_DISK in the HRB. Finally, it stores the starting LBN in the IRP and queues the IRP to the specific driver for the device.

Just as in the READ case, the driver handles the IRP as if the request had originated on the local node. The data is written to the disk from the transfer buffer, and eventually I/O postprocessing is invoked. *IOC$IOPOST* passes the IRP to routine BACK, which in turn resumes the request within the MSCP server at the instruction in routine WRITE following the call to DO_DISK.

Routine WRITE updates the Accumulated Byte Count field in the HRB and compares it against the HRB's Original Byte Count" field. If more bytes remain to be written to the disk, then WRITE branches back to initialize the CDRP to request more data from the remote host. If the WRITE request is complete, then an MSCP End Message is sent to the remote host indicating that the transfer was successful.

5.6.3 Command Status

DUDRIVER maintains CDRPs representing commands that are active for a controller in a queue attached to the CDDB for that controller. The queue is ordered such that the older a command is, the closer it is to the head of the queue.

Once every controller timeout interval, DUDRIVER's *Timeout Mechanism* (described in the preceding chapter) compares the RSPID in the CDDB with the RSPID in the CDRP at the head of the queue.

- If they are different, then the former oldest command has been completed, another CDRP has advanced to the head of the queue, and the command represented by this other CDRP has thus assumed the role of being the oldest. In this case, the RSPID is copied from the CDRP at the head of the queue into the CDDB's OLDRSPID field, and the CDDB$L_ OLDCMDSTS field is set to -1.

- If they are the same, then the command at the head of the CDRP queue has been the oldest active command for at least one controller timeout interval. Suspicious that the controller may be "very ill", DUDRIVER issues a GET COMMAND STATUS to the controller, inquiring if the controller has made any progress on this command.

This applies not only to a DSA controller, but also to a VMS based MSCP server which is emulating a DSA controller. The VMS based MSCP server must keep track of whether or not it is making progress on a READ or WRITE command it receives from each remote VAX host.

When the MSCP server initializes an HRB for a command received from a remote host, it sets the *CMD_STS* field to negative 2. It then decrements the CMD_STS field at strategic points within its processing of MSCP READ and WRITE commands.

- For a READ command, this happens immediately following the point where the request resumes after transferring the data to the remote host, and just before adding the number of bytes read into the HRB's accumulated byte count field.
- For a WRITE command, this happens immediately after the data to be written to the disk is received from the remote host, and just before preparing the IRP to be handed to the local host's driver for the disk.
- For both a READ and a WRITE request, the CMD_STS field is decremented in the special I/O postprocessing routine BACK, just before routine BACK resumes the request following the call to DO_DISK.

When the MSCP server receives a GET COMMAND STATUS, it scans the queue of HRBs attached to the HQB corresponding to the host which sent the GET COMMAND STATUS. Finding the HRB containing a command reference number matching the one in the GET COMMAND STATUS, it copies the content of the HRB$L_CMD_STS field into the GET COMMAND STATUS. Finally, it converts the GET COMMAND STATUS into a corresponding MSCP end message, and sends the end message containing the command status back to the remote host.

NOTE

The command reference number in the MSCP packets is the RSPID allocated for the I/O request by DUDRIVER in the remote host.

The remote host compares the command status in the end message with the OLDCMDSTS field in the CDDB. If the command status in the end message is numerically less than the content of the OLDCMDSTS field, then progress has been made by the server on the command. The command status in the end message is copied into the OLDCMDSTS field just in case the timeout mechanism finds the same oldest active command the next time it is invoked, and thus has to issue another GET COMMAND STATUS for it.

On the other hand, if the command status in the message is not less than the content of the OLDCMDSTS field, then the server has failed to make progress on the command in a reasonable amount of time. The remote DUDRIVER presumes it to be "very ill" and attempts to resynchronize activity with the server.

5.6.4 Details of the Routines for Handling READ and WRITE Commands

The next few sections detail how the VMS based MSCP server handles MSCP READ and WRITE commands received from remote hosts.

5.6.4.1 MSG_IN - Receiving Command and Server Resource Allocation

Routine MSG_IN verifies that the connection is not being broken, and then increments the operation count in the Disk Server Structure. It then validates the MSCP command, allocates an HRB and IRP to represent the request, and dispatches to the appropriate routine based on the class of the MSCP command.

- Calls *ALLOCATE_HRB* to allocate an HRB and IRP/CDRP and initializes selected fields in these data structures.
  - — An HRB is allocated from nonpaged pool.
  - — Various HRB fields are set to zero: State, Flags, Respc, Msgbuf, Irp_Cdrp, Bufadr, Svapte, Boff, Bcnt, Abcnt, Uqb.
  - — The CMD_STS field in the HRB is set to negative 2.
  - — An IRP/CDRP pair is allocated from nonpaged pool and its address is stored in the IRP_CDRP field of the HRB.
  - — Various IRP fields are set to zero: Rmod, Wind, Efn, Chan, Sts, and Rspid.
  - — The address of the HRB is stored in the IRP.
  - — The CDRP fields FQFL and RWCPTR (Rwaitcnt Pointer) are set to zero, and the *Fork Lock Field* (FLCK) is set to contain *SPL$C_SCS*.
- The *STATE_INVALID* flag is set in the HRB$W_STATE field, making the HRB the Current Request.
- The HQB address is copied from the *CDT$L_AUXSTRUC* offset in the CDT into the *HRB$L_HQB* offset of the HRB.
- Store the address of the beginning of the MSCP command buffer passed in the SCS message in the HRB at offset HRB$L_MSGBUF.
- The HRB is queued to the HQB associated with the host which issued the MSCP command.
- The Opcode, length, modifiers, and flags fields are validated. If any are invalid, an MSCP End Message is sent to the disk class driver on the remote host with the appropriate MSCP error code, and no further processing is done for this request.
- MSG_IN dispatches based on the "class field" within the MSCP opcode.

NOTE

READ and WRITE commands are classified as *NonSequential Buffered* commands. The routine to which dispatch is made is *NONSEQB*.

5.6.4.2 NONSEQB - Verifying that Command Processing may Continue

The NONSEQB routine verifies that the disk is online to the requesting host and that no *Sequential Commands/italic* for the disk are either in progress or pending. It none are found, it then dispatches based on the specified opcode (READ or WRITE).

- Calls routine *FIND_UQB* to get the address of the Unit Queue Block corresponding to the disk unit for which the MSCP command is intended.
  - — Fetches the address of the HQB from the HRB, and then the address of the DSRV from the HQB.
  - — A Check is made to see if the MSCP$W_UNIT field contains the value of *MSCP$K_SLUN_RSVP* indicating that this is the first time that a particular Class Driver has requested a Get Unit Status for this device since its connection to the server was formed.
    - o If this is an RSVP, the list of UQBs is scanned for a *Unit Identifier* that matches the requested unit. If found, the *Server Local Unit Number* corresponding to this unit will be returned to the requesting host in the MSCP$W_UNIT field of the MSCP packet for future unit referencing. An OFFLINE error will be returned if the unit is not located.
    - o If this is Not an RSVP, the Server Local Unit Number is already known and has been passed in the unit field of the MSCP packet.
  - — The SLUN is used to index into the *DSRV$L_UNITS* vector to locate the UQB address. The address of the UQB is then stored in the HRB.

NOTE

If no matching UQB is found, an end message with OFFLINE status and a "device unknown" subcode is sent to the disk class driver on the remote host, and no further processing for this request is performed.

- Verifies from the UQB that the unit is online to the requesting host.

NOTE

If not, an end message with MSCP$K_ST_AVLBL status is sent to the class driver on the remote host, and no further processing is performed for this command.

- Verifies that no sequential commands are pending, and that no requests are in the blocked command queue:
  - — SEQ bit in UQB$W_FLAGS not set.
  - — UQB$W_NUM_QUE field contains zero.

NOTE

A request being blocked indicates the presence of a sequential command, either pending or executing.

If either is true, the *HRB$W_STATE* field is set to contain *HRB$K_ST_SEQ_WAIT* (this command is also blocked), and the HRB is inserted at tail of the UQB blocked queue. Effectively, this request is suspended here until it unblocks.

- NONSEQB dispatches based on the MSCP op code (READ or WRITE).

5.6.4.3 READ - Processing MSCP READ Command

Allocates transfer buffer and SCS mapping resources, prepares the IRP for disk transfer, executes disk transfer by passing IRP to appropriate driver, sends data to remote host, and finally sends MSCP end message to remote host.

- Branches to subroutine ALLOCATE to allocate a transfer buffer for data to be read from the disk.
  - — Copies the total size of the transfer from the MSCP command to the OBCNT field in the HRB.
  - — Copies the starting LBN from the MSCP command to the HRB$L_LBN field in HRB.
  - — If the requested transfer size is greater than 127 blocks, use 127 blocks for this segment
  - — Allocates a transfer buffer from the "local" nonpaged transfer buffer pool pre-allocated to the MSCP server during its initialization.

NOTE

Offset DSRV$L_FREE_LIST in DSRV contains the address of the beginning of the list of free pool fragments.

If the total number of bytes to transfer, HRB$L_OBCNT, exceeds the largest number of bytes permitted in a single I/O transfer, (the maximum of one half the available pool or 512) then the requested buffer size is constrained to this limit.

An Attempt is made to allocate the buffer. If no pool fragment is large enough, the requested size is divided by two and another attempt is made to allocate a fragment. This is repeated until the allocation succeeds or until the requested size is reduced to the point where it meets the following criteria:

- o Requested size is less than one fourth the total size of the transfer.
- o Requested size is less than the content of the *DSRV$L_BUFFER_MIN* field in the server's DSRV (one eighth the total buffer pool)

If the allocation fails, the HRB is placed in the *HRB$K_ST_BUF_WAIT* state and queued to the *DSRV Memory Wait Queue* (DSRV$L_MEMW_BL) .

  - — The actual length and address of the transfer buffer is stored in the HRB at offsets *HRB$L_BUFLEN* and *HRB$L_BUFADR*, respectively.

  - If the buflen returned is less than the original byte count, increment the split transfer field (DSRV$L_SPLITXFER) in the DSRV.
  - The lesser of BUFLEN and OBCNT in the HRB is copied to the BCNT field in HRB.
  - The byte offset (BOFF) is set in the HRB.
  - The system virtual address of the system PTE pointing to this buffer is stored in the *HRB$L_SVAPTE* field of the HRB. (MMG$GL_SPTBASE indexed by the Virtual Page Number)
- Copies the address of the CDT from the HQB into the CDRP.
- Sets the state of the HRB to *HRB$K_ST_MAP_WAIT*.
- Since data is to be read from a disk into the newly allocated buffer and then transferred from the buffer to the remote host that issued the READ command, this transfer buffer must be mapped. The data will be sent to the remote host by means of the local host's CI, DSSI or NI port. Mapping resources specific to PADRIVER, PIDRIVER or PEDRIVER (whose top layer emulates PADRIVER) will be used for this purpose. The port dependent Buffer allocation routines will be called by macro *MAP* to perform the following:
  - Removes a free buffer descriptor from the linked list in the BDT and initializes the buffer descriptor based on the SVAPTE, BCNT, and BOFF fields in the HRB.
  - Initializes the buffer handle at offset HRB$L_LBUFF in the HRB. (Transfer offset = zero, buffer name = BDT index, RCONID from CDT.)

NOTE

If no free buffer descriptor is available, the CDRP is queued to the BDT wait queue and the request is suspended at this point until a free buffer descriptor becomes available.

- Makes the HRB "current" again by setting HRB$M_ST_INVALID flag.
- Sets the MAP flag in HRB$W_FLAGS field, indicating mapping resources have been allocated for this request.
- Prepares the IRP for the actual disk transfer.
  - The address of the UCB is copied from the UQB into the IRP.
  - SVAPTE, BOFF, and BCNT fields of HRB copied into the IRP.
  - FUNC field of the IRP is set to indicate a READ operation (IO$_READPBLK).
- Calls DO_DISK to execute the disk transfer.
  - Verifies from the UQB that the unit is online relative to the remote host. (If it isn't, an end message with SS$_MEDOFL status is sent to the remote host.)
  - The address of where to return after I/O postprocessing is popped from the stack into the *HRB$L_RESPC* field. (This is the address of the instruction immediately following the call to DO_DISK.)
  - The address of the *I/O Postprocessing Routine, BACK,* is stored in the IRP$L_PID field.
  - Calls *IOC$CVTLOGPHY*, which stores the starting LBN in the MEDIA field of the IRP.
  - Sets the state of the HRB to HRB$K_ST_DRV_WAIT.

— Branches to EXE$INSIOQ to hand off the IRP to the driver's STARTIO routine as if the IRP came from a locally issued $QIO.

5.6.4.4 IOC$IOPOST - I/O Postprocessing for READ

Invoked by *IPL$_IOPOST* software interrupt requested by the driver after actually executing the disk transfer. The driver has already constructed and stored IOSB information in the IRP. I/O postprocessing is responsible for resuming this request at the address stored in the RESPC field of the HRB. For a READ operation, this is the address of the instruction immediately following the call to DO_DISK made from routine READ.

- Examines the IRP$L_PID field and finds that it does not contain a PID, but rather the address of an "end action" routine, namely BACK. Routine BACK is called.
- Routine BACK actually resumes the request.
  - Sets the INVALID flag in the STATE field of the HRB, making it the "current" HRB.
  - Calls the routine whose address is at offset RESPC, resuming this request immediately following the call to DO_DISK.

5.6.4.5 Read Request Resumes Following DO_DISK

Calls routine to send data just read from disk to remote host, updates accumulated byte count. Increments the operation count in the HULB. If entire request not yet satisfied, branches back to read more data from disk. If entire request is satisfied, branches to routine to send end message to remote host.

- The number of bytes read from the disk is copied from the BCNT field in the IRP to the BCNT field in the HRB.
- Increments the Operation Count in the HULB structure by locating the HULB_VECTOR in the HQB and indexing by the Server Local Unit Number.
- Initializes the CDRP so that it may be used by SCS to send retrieved data to remote host.
  - The CDT address is copied into the CDRP from the HQB.
  - The number of bytes transferred is copied from the BCNT field in the HRB to the XCT_LEN field in the CDRP.
  - Address of remote buffer handle in original MSCP READ command copied into RBUFH_AD field of CDRP.
  - The offset into the remote host's buffer (i.e. current content of HRB$L_ABCNT field) is copied to the RBOFF field in the CDRP.
  - Defines local buffer handle to be the buffer handle in the HRB by setting the CDRP$L_LBUFH_AD field to contain the address of offset LBUFF in HRB.
  - Clears the CDRP$L_LBOFF field.
- If this is the last transfer required, routine SEND_DATA_WMSG will be used to piggyback and MSCP End message onto the final transfer.

- If this is not the last transfer required, set the state of the HRB to *HRB$K_SNDAT_WAIT*, and then call routines to send the data read from the disk to the remote host. The CDRP passed to these routines is the CDRP initialized above to reference the mapped transfer buffer containing the data.

  Table 5–6 lists the routines involved in sending the data to the remote host.

Table 5–6: Routines Invoked by the SEND_DATA Macro

Routine	Description
SCS$ALLOC_RSPID	Allocates a RSPID and associated RDT entry, stores RSPID in CDRP.
FPC$ALLOCMSG	Allocates message buffer in which to build SNDDAT command, and saves address of this buffer in CDRP.
FPC$SENDDATA	Builds and queues to the port a SNDDAT command to transmit data read from disk to remote host.

NOTE

The Fork Thread is suspended with context saved in the CDRP until the transfer of data to remote host completes. The address of where to resume this fork thread is saved in CDRP at offset FPC. When the last packet containing data from the disk is received by the remote host, its port generates a confirmation packet containing the RSPID.

When the confirmation is received by the local host, its port driver uses the RSPID to find the CDRP and resumes this fork thread. In so doing, the RSPID and RDT entry are released. SCS or pool waits may also occur if no free RSPID and RDT or pool are available when the allocation of these resources is attempted.

- The request resumes after the transfer of data read from the disk is complete. The HRB state is set to "current" by setting the INVALID flag in the HRB.
- Indicate progress being made for this command (Decrement CMD_STS)
- The number of bytes just sent to the remote host is added into the accumulated byte count field, ABCNT, in the HRB. The content of the ABCNT field is subtracted from content of OBCNT field in the HRB to determine if more bytes remain to be read from the disk and sent to the remote host.
- If more bytes remain,
  - — The lesser of the number of bytes remaining to be read and the current content of the HRB$L_BCNT field is stored in the HRB$L_BCNT field.
  - — HRB$L_LBN is updated by adding to it the number of blocks just sent to the remote host so that it points to the next LBN to be read from the disk.
  - — Branches back to prepare the IRP for the next actual disk transfer.

- If all desired data has now been read from the disk and sent to the remote host, a special SEND_DATA will be used to piggyback an MSCP End message onto the last block transfer.
  - — The content of ABCNT field in the HRB is copied into the BYTE_CNT field of the buffer containing the original MSCP READ command.
  - — Branches to routine *SEND_DATA_WMSG*, passing it the status code MSCP$K_ST_SUCC indicating success.
- Branches to routine *CLEANUP_HRB* to deallocate the resources held by the HRB.

5.6.4.6 WRITE - Processing MSCP WRITE Command

Allocates transfer buffer and SCS mapping resources, requests data from host wishing to write data to disk, Increments the Operation Count in the HULB, prepares an IRP for the disk transfer, executes the disk transfer by passing IRP to appropriate driver, and finally sends an MSCP end message to the host which issued WRITE request.

- Verifies that the disk is neither *Hardware nor Software Write Protected* by testing the UF_WRTPH and UF_WRTPS bits in the UNIT_FLAGS field of the UQB.

NOTE

If either is true, then an end message is sent with an MSCP$K_ST_WRTPR status code.

- Calls ALLOCATE to allocate a transfer buffer to hold the data which is to be written to the disk.

NOTE

ALLOCATE performs the same tasks here for an MSCP WRITE command as it does when called for an MSCP READ command. See Section 5.6.4.3 for details.

- Copies the address of the CDT from the HQB into the CDRP.
- Sets the state of the HRB to HRB$K_ST_MAP_WAIT.
- Maps newly allocated buffer by calling the *MAP* macro.

NOTE

The same tasks are performed here as for the MSCP READ command described in Section 5.6.4.3. The main difference is that the mapping is for purposes of transferring data from the remote host to the transfer buffer. The same mapping resources will be used since the same types of ports are involved as with the READ command.

- Sets the HRB$M_STATE_INVALID flag in the HRB$W_STATE field, thereby making this HRB the "current" HRB.

- Sets the MAP flag in the HRB$W_FLAGS field, indicating that mapping resources have been allocated for this request.
- Increments the Operation Count field in the HULB structure by locating the HULB_ VECTOR in the HQB and indexing by the Server Local Unit Number.
- Initializes the CDRP so that it may be used by SCS to send data from the system which issued the MSCP WRITE command.

NOTE

The same tasks are performed here as would be performed for initializing the CDRP while handling a READ command to send data retrieved from disk to the remote host. See Section 5.6.4.3 for detail.

- Sets the state of the HRB to HRB$K_ST_SNDAT_WAIT.
- Calls SCS routines to request that the host which issued the WRITE command send data to be written to the disk by invoking the REQUEST_DATA macro.

  Table 5–7 lists the routines involved in requesting the data from the remote host.

Table 5–7: Routines Invoked by the REQUEST_DATA Macro

Routine	Description
SCS$ALLOC_RSPID	allocates a RSPID and associated RDT entry, stores RSPID in CDRP.
FPC$ALLOCMSG	allocates message buffer in which to build SNDDAT command, saves address of this buffer in CDRP.
FPC$REQDATA	builds and queues to port REQDAT command to request data from host which issued MSCP WRITE command.

NOTE

The Fork Thread is suspended with context saved in the CDRP until the transfer of data to the serving host completes. Address of where to resume the fork thread is also saved in CDRP.

When the last packet containing data to be written to the disk is received, the local port generates a DATREC containing the RSPID. The local port driver uses the RSPID to find the CDRP and resume this fork thread. In doing so, the RSPID and associated RDT entry are released.

- When the request resumes, the HRB state is set to "current" by setting the the HRB$M_ STATE_INVALID flag.
- Prepares the IRP for actual disk transfer.
  - — Copies the address of the UCB from the UQB into the IRP.
  - — SVAPTE, BOFF, and BCNT fields copied from HRB to IRP.
  - — Sets FUNC field of IRP to indicate a WRITE operation (IO$_WRITEPBLK).

- Calls DO_DISK to execute the disk transfer.

NOTE

The same tasks are performed here as would be performed by DO_DISK while handling a READ command. See Section 5.6.4.3 for details.

5.6.4.7 IOC$IOPOST - I/O Postprocessing for WRITE

Invoked by IPL$_IOPOST software interrupt requested by the driver after actually writing the data to the disk. The driver has already constructed the IOSB information and stored it in the IRP. I/O postprocessing resumes this request at the address stored in RESPC field of the HRB. For a WRITE operation, this is the address of the instruction immediately following the call to DO_DISK made from routine WRITE.

The steps here are the same as for a READ command since the IPL$_IOPOST software interrupt is always serviced by routine IOC$IOPOST:

- Examines the IRP$L_PID field and finds it does not contain a PID, but rather the address of an "end action" routine, namely BACK. Routine BACK is called.
- Routine BACK actually resumes the request.
  - — Sets the INVALID flag in the STATE field of the HRB, making this HRB the "current" HRB.
  - — Calls the routine whose address is at offset RESPC, resuming this request immediately following the call to DO_DISK.

5.6.4.8 Write Request Resumes Following DO_DISK

Updates accumulated byte count. If the request is incomplete due to fragmentation, branches back to request more data to be written to disk from the remote host. If the request is now complete, branches to a routine which issues the MSCP End message to the remote host.

- Number of bytes just written to disk, IRP$L_BCNT, is added into accumulated byte count field, ABCNT, in the HRB. The content of the ABCNT field is subtracted from the content of the OBCNT field in the HRB to determine if more bytes remain to be written to the disk for this request.
- If more bytes remain:
  - — The lesser of two quantities, the number of bytes remaining to be written to disk and the current content of HRB$L_BCNT field, is stored in the HRB$L_BCNT field.
  - — HRB$L_LBN is updated by adding to it the number of blocks just written to the disk so that it points to the next LBN to be written.
  - — Branches back to initialize the CDRP so that the CDRP may be used by SCS to request still more data from the remote host which issued the MSCP WRITE command.

- If all data for this request has now been written to the disk:
  - The content of the ABCNT field in the HRB is copied to the BYTE_CNT field of the buffer containing original MSCP WRITE command.
  - Branches to routine *SEND_END*, passing it the status code MSCP$K_ST_SUCC indicating success.

5.6.4.9 SEND_END - Send End Message and Cleanup

This routine is branched to for the completion of WRITE requests. It converts the received MSCP command into an MSCP End message, sends the MSCP End message to the host which issued the command, and deallocates any resources still held by the HRB.

- Stores the status code into the MSCP$W_STATUS field of the buffer containing the original MSCP command.
- Alters the MSCP$B_OPCODE field in the message buffer so that it reflects an MSCP End message corresponding to the original command issued by the remote host. This is done by merely setting the OP_END bit within this field.
- Initializes the CDRP in preparation for requesting SCS to send the end message.
  - — The address of the CDT is copied into the CDRP from the HQB.
  - — The address of the message buffer is stored in the CDRP.
- Set the HRB state to *HRB$K_ST_MSG_WAIT* and the message buffer is recycled in preparation for sending it back to the remote host as an end message by invoking macro *RECYCL_MSG_BUF*.
- The Recycle Message Buffer routine performs the following:
  - — CDT checked to see if at least one send credit available. (If not, this fork thread is suspended until one is available.)
  - — RCONID copied from CDT into DST_CONID of SCS header in message buffer.
  - — Number of send credits decremented by 1.
- HRB is made "current" by setting the INVALID flag in its STATE field.
- HRB state is then set to *HRB$K_ST_SNDMS_WAIT*.
- Calls *FPC$SNDCNTMSG* to pass the buffer containing the MSCP End message to the SCS layer for transmission to the host which issued the original MSCP command.
- Calls CLEANUP_HRB to deallocate (or at least release) any resources held by the HRB (IRP/CDRP, local buffer for holding data being transferred between the disk and the remote host, SCS mapping resources, etc.), and deallocates the HRB itself.

5.7 Other Classes of Commands Handled by the Server

5.7.1 Overview

There are three additional classes of MSCP commands that are handled by the VMS based MSCP server: *Immediate Commands, NonSequential Nonbuffered Commands*, and *Sequential Commands.*

5.7.1.1 Immediate Commands

An immediate command requires very little time to complete, does not cause any unit to experience a context change, and must be processed immediately by an MSCP server. DUDRIVER expects an MSCP server to complete processing of an immediate command and return an MSCP End message for that command within one controller timeout period after the command is issued. If DUDRIVER does not receive the end message within this time frame, it will attempt to resynchronize its activity with the server.

This involves breaking the SCS connection between itself and the server. If the "MSCP speaking" controller in which the server resides is not a VAX emulating a controller, then this also involves issuing a *Host Clear* to reset the controller.

NOTE

The controller timeout period was established when the SCS connection formed between the driver and the server.

Two examples of immediate commands are

- ABORT, to abort an MSCP command previously issued to the controller, but for which the controller has not yet sent to the issuing host a corresponding end message.
- GET COMMAND STATUS, to determine if the controller has made any progress on an outstanding MSCP command issued to the controller.

Through the use of SCS flow control, the disk class driver is always assured that it has at least one send credit to issue an immediate command to an MSCP server. This is accomplished by requiring that DUDRIVER's connection with an MSCP server always have at least two send credits before being allowed to send any other type of command to the server. Only one send credit has to be available to send an immediate command.

Of particular concern here is the desirability of guaranteeing that DUDRIVER can always issue a *GET COMMAND STATUS* and have it complete within the controller timeout interval.

5.7.1.2 NonSequential Commands

NonSequential commands are those commands that a controller may re-order for reasons of performance optimization. Furthermore, they may be segmented, and then performed one segment at a time. The execution of two or more NonSequential commands may be interleaved by interleaving the execution of their segments.

Read and write requests are examples of NonSequential Buffered commands. The controller must allocate buffers to hold the data while it is being exchanged between a unit and a host. NonSequential Nonbuffered commands require no such buffers. Three examples of NonSequential Nonbuffered commands are:

- ACCESS, to verify that designated data on some unit can be read without errors.
- ERASE, to overwrite a region of a unit with zeros.
- REPLACE, to mark a logical block as being replaced by a replacement block.

The VMS based MSCP server does not handle all immediate, sequential, and NonSequential nonbuffered commands. The next few sections explain which commands in each category it does, and to what extent.

5.7.1.3 Sequential Commands

Sequential commands are commands that, as the name implies, must be executed in a precise order. Furthermore, they impact the execution of other types of commands. There are three rules that define the precise order in which sequential commands can be executed, and how they impact other commands:

- All sequential commands received on the same SCS connection for a particular unit must be executed in the exact order in which the server receives them.
- NonSequential commands received on the same connection for the same unit as a sequential command, but before that sequential command, must be completed before executing the sequential command.
- NonSequential commands received on the same connection for the same unit as a sequential command, but after the sequential command, must not begin execution before the sequential command completes.

Because of these rules, sequential commands are "well ordered", and each one effectively forms a barrier around or through which NonSequential commands may not pass. The reason for this is that sequential commands typically alter the context of a unit. To illustrate this, here are three examples of sequential commands:

- AVAILABLE, to place a unit in the "available" state relative to the host that issued the command (and relative to all hosts which are clients of the MSCP server if none of them still have the unit online).
- ONLINE, to set the unit "online" relative to the host which issued the command.
- SET UNIT CHARACTERISTICS, to establish host settable unit characteristics.

5.7.2 Immediate Class Commands

The VMS based MSCP server supports four immediate class commands sent to it by a host:

Table 5–8: Supported Immediate Class Commands

Command	Application
ABORT	to abort a command being handled by the server
GET COMMAND STATUS	to obtain the disposition of a command
GET UNIT STATUS	to request the status of a unit which is accessible via the server
SET CONTROLLER CHARACTERISTICS	to establish host settable controller characteristics

All immediate class commands are passed to routine *IMMEDIATE* by the MSCP server's SCS message input routine MSG_IN. IMMEDIATE then dispatches each of these commands to a command specific routine for processing. Here is a summary of each of the four command specific routines:

- ABORT

  All MSCP commands from a particular host are represented by HRBs in a queue attached to the HQB for that host. This queue is searched for an HRB whose command reference number and unit number fields match those in the ABORT request. This HRB is marked as "aborted" by setting the *ABORTWS* bit in its FLAGS field.

  The STATE field of the HRB for the command to be aborted indicates at what stage of processing the CDRP is for that command, and how the command is actually to be aborted.

  If the CDRP for the command being aborted has not yet been handed to a driver, and if it is not waiting to be allocated SCS mapping resources, then it is under the "jurisdiction" of the MSCP server. The CDRP (and attached IRP) are released, along with any resources (buffers, RSPID, RDT entry, etc.) held by the HRB. The HRB itself is then deallocated.

  If the CDRP for the command being aborted is currently being handled by a driver, or if it is waiting to be given SCS mapping resources buffer, then it is not currently under the "jurisdiction" of the MSCP server. Eventually it will be returned to the server. When this occurs, the ABORTWS flag being set in the HRB will trigger the release of the CDRP (and IRP) and resources held by the HRB, and the deallocation of the HRB.

  Associated with the ABORT command will be an end message indicating that the command referenced by the ABORT command has been successfully aborted (MSCP major status code = MSCP$K_ST_SUCC). But an end message will still be sent to the host for the command which was aborted (MSCP major status code = MSCP$K_ST_ABORTD).

  NOTE

  The ABORTWS flag discussed here should not be confused with the ABORT flag discussed in a later section presenting how the MSCP server handles the loss of the SCS connection with the remote disk class driver. ABORTWS stands for *Abort With Status* since an end message corresponding to the aborted command,

and containing the MSCP$K_ST_ABORTD status code, is sent to the remote host.

When an HRB is terminated due to SCS connection failure, the ABORT flag is set in the HRB; this indicates that the HRB is to be terminated without status being being sent to a remote host. If the SCS connection is broken, there is no remote host to which an end message can be sent.

- GET_COMMAND_STATUS

  The queue of HRBs attached to the HQB which issued the GET COMMAND STATUS is searched for an HRB whose Command Reference Number and unit number fields match those in the GET COMMAND STATUS request. If the desired HRB is found, then a copy of the content of the CMD_STS field of the HRB is sent to the requesting host in an MSCP End message. If the desired HRB is not found, the command status field in the End message sent to the requesting host is set to 0.

- GET_UNIT_STATUS

  The purpose of the GET UNIT STATUS command is to obtain the current status of a unit, and certain unit characteristics. Some of the items returned in the end message for such a command are:

  — MSCP unit number.

  — Media ID.

  — Disk geometry.

  — Unit identifier.

  — Status code: Available, Offline, or Online.

  The GET UNIT STATUS command can request this information for a specific unit. It can also request status for the next unit known to the server with a unit number equal to or greater than the specified unit by specifying the *MSCP$V_MD_NXUNT* flag in the MSCP$W_MODIFIER field.

  Routine GET_UNIT_STATUS obtains this information from the UQB corresponding to the unit specified in the command. The UQB is found by using the SLUN to index into the unit table of the DSRV.

  If no matching UQB is found, a device offline is returned to the requesting node. The requesting node can then look for an alternate path.

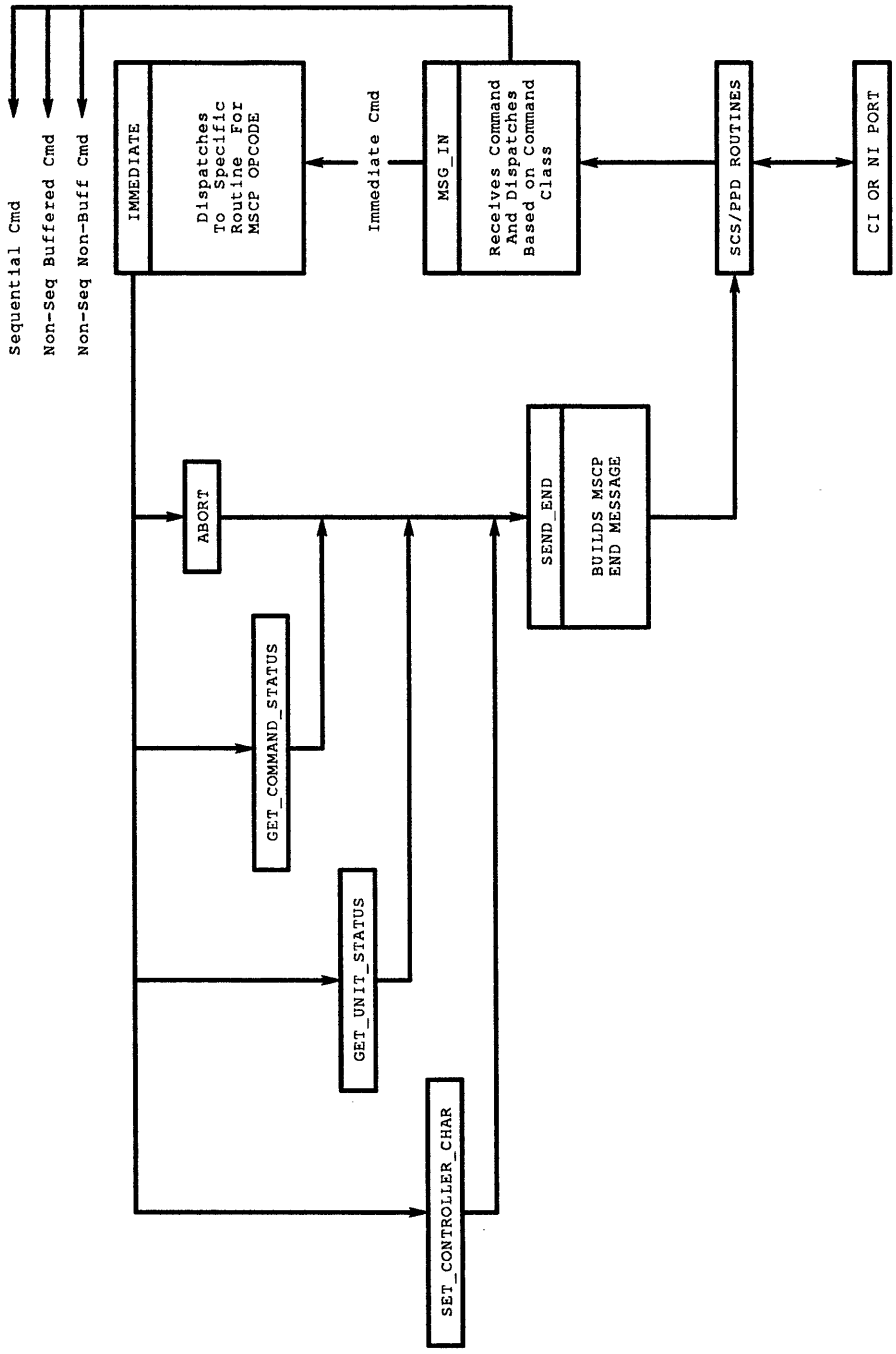
- SET_CONTROLLER_CHAR

  This command is the mechanism by which certain host settable characteristics are determined for a controller. The MSCP server will present to a host the "illusion" of taking on these characteristics since it is emulating an "MSCP speaking" controller.

  Host settable characteristics include controller flags (e.g. enable attention messages, enable miscellaneous error log messages, ...) and the host access timeout period. These characteristics are stored in the HQB corresponding to the remote host which issued the SET CONTROLLER CHARACTERISTICS command. The VMS based MSCP server also stores the local date and time in the HQB at offset *HQB$Q_TIME* when this command is received.

The end message returned to the remote host includes the MSCP server's allocation class, controller software version number, controller flag settings, and controller timeout interval. It also includes a controller identifier which identifies itself to the host as a VMS system emulating an "MSCP speaking" controller by means of the VMS based MSCP server. Figure 5–17 Illustrates the general flow for Immediate Class Commands.

Figure 5–17: General Flow of Immediate Class Commands



CXN-0005-02

5.7.2.1 Routines for Handling Immediate Commands

Following are detailed outlines of the four command specific routines for handling immediate class commands.

Routine IMMEDIATE passes the addresses of the immediate MSCP command and the HRB corresponding to the host which sent the command to each routine.

5.7.2.1.1 ABORT

Aborting a command begins with seeking out the HRB representing that command, and marking the HRB as "aborted". If the IRP/CDRP pair for the command is still within the jurisdiction of the MSCP server, the IRP/CDRP pair along with the HRB and any resources held by the HRB are released/deallocated. If the IRP/CDRP pair are not within the jurisdiction of the MSCP server, then these data structures and resources will be released as soon as possible.

- Routine ABORT begins by finding the HRB representing the command that is to be aborted. It fetches the HQB corresponding to the host which issued the ABORT, and then searches the queue of HRBs attached to the HQB for an HRB with a command reference number and unit number matching those contained in the ABORT.
  - — If no matching HRB is present in the queue, then ABORT branches to SEND_END to issue a "success" End message to the host which sent the ABORT command. No further processing is done for the ABORT request.
  - — If a matching HRB is found, then the command corresponding to the HRB is marked as being aborted by setting the ABORTWS bit in the FLAGS field of the HRB.
- The HRB representing the request to be aborted is now handled according to to its state (HRB$W_STATE field).

  - — HRB$K_ST_MSG_WAIT - Waiting for an SCS message buffer or send credit.

    Return Success status back for the Abort Request End message.

  - — HRB$K_ST_SEQ_WAIT - Waiting for completion of a sequential command.

    The HRB is removed from the HQB's blocked command queue, and the *UQB$W_NUM_QUE* field in the UQB is decremented. The STATE_INVALID flag is set in the state field of the HRB. The request is marked as DEQUEUED in the flags field of the HRB. An Aborted status is set for the End message. The accumulated byte count is set to reflect any transfers that had succeeded.

  - — HRB$K_ST_BUF_WAIT - Waiting for local MSCP server buffer.

    The HRB is removed from the queue of commands waiting for transfer buffers, and the *DSRV$W_MEMW_CNT* field is decremented. The STATE_INVALID flag is set in the state field of the HRB. The request is marked as DEQUEUED in the flags field of the HRB. An Aborted status is set for the End message. The accumulated byte count is set to reflect any transfers that had succeeded.

— HRB$K_ST_SNDAT_WAIT - Waiting for completion of a block data transfer.

The request is currently suspended. The RSPID wait queue (RDT$L_WAITQFL) is scanned for the CDRP and Unhooked if found. The Message Buf wait queue is then scanned looking for the request, and if found it is Unhooked. The CDRP is removed from the wait queue, or the RDT entry is released. Success status is returned back for the Abort Request End message.

— HRB$K_ST_DRV_WAIT - Waiting for driver to complete request.

The request's IRP has already been queued to the driver, so it is currently outside the "jurisdiction" of the MSCP server. However, the fact that the ABORTWS flag has been set in the HRB will cause the command to be aborted when the IRP is handed back to the server by I/O postprocessing.

This routine merely branches to SEND_END to transmit a "success" End message to the remote host that issued the ABORT request. It should be observed that this End message corresponds to the ABORT request, and not the MSCP command being aborted. A separate "aborted" End message will be sent when the command being aborted actually is.

— HRB$K_ST_MAP_WAIT - Waiting for SCS mapping resources.

The request is waiting for SCS mapping resources. So it is currently outside the "jurisdiction" of the MSCP server. However, the fact that the ABORTWS flag has been set in the HRB request will cause it to be aborted when the IRP is handed back to the server by the mapping routine.

This routine merely branches to SEND_END to transmit a "success" End message to the remote host that issued the ABORT request. It should be observed that this End message corresponds to the ABORT Request, and not the actual MSCP command which is being aborted. A separate "aborted" end message will be sent when the command being aborted actually is.

- Routine ABORT then calls SEND_END to transmit to the remote host an end message bearing the status code MSCP$K_ST_ABORTD. This end message corresponds to the aborted MSCP command.

  SEND_END will also release any resources held by the aborted HRB (such as SCS mapping resources, RDT entry and RSPID, local or SCS buffers, ...), and then deallocate the HRB itself.
- Upon return from the first call to SEND_END, routine ABORT again branches to SEND_END to transmit a "success" End message for the Abort command itself.

5.7.2.1.2 GET_COMMAND_STATUS

This routine locates the HRB corresponding to the command for which status has been requested. The content of the command status field of the HRB is sent to the requesting host in an End message.

- Routine GET_COMMAND_STATUS begins by fetching the HQB corresponding to the requesting host. It then searches the queue of HRBs attached to the HQB, looking for an HRB satisfying two conditions:
  - — The command reference number in the message buffer attached to the HRB matches the command reference number in the buffer containing the GET COMMAND STATUS request.
  - — The unit number in the message buffer attached to the HRB matches the unit number in the buffer containing the GET COMMAND STATUS request.
- If an HRB matching both these conditions is found, then the content of the command status field of the message buffer attached to the HRB is copied into the command status field of the buffer containing the GET COMMAND STATUS request.

  If not, then the command status field of the buffer containing the GET COMMAND STATUS request is set to zero.
- GET_COMMAND_STATUS then branches to SEND_END, passing it an MSCP major status code of MSCP$K_ST_SUCC (success). SEND_END will then
  - — Store the MSCP major status code in the buffer containing the GET COMMAND STATUS request.
  - — Convert the received GET COMMAND STATUS into an MSCP end message.
  - — Send the end message to the requesting host.

5.7.2.1.3 GET_UNIT_STATUS

If the "next unit" modifier is set, the UQB with the first unit number greater than or equal to the unit number in the GET UNIT STATUS command is sought; otherwise, the UQB corresponding to the command's unit number is found. Status information is extracted from the UQB and returned in an end message to the requesting host.

- First, GET_UNIT_STATUS selects the proper UQB from which to extract the information that is to be returned to the requesting host.
  - — If the "next unit" modifier (MSCP$V_MD_NXTUNT) is set in the MODIFIER field of the request, then the Server Local Unit Number is used to index into the list of units kept in the DSRV for the first index that points to a valid UQB.

    NOTE

    The queue of UQBs is ordered in ascending order by unit number.

    - o If none is found, then
      - — The MSCP$W_UNIT field in the request is cleared to zero.
      - — The Status OFFLN is returned in the MSCP End message.

    - o If a UQB is found satisfying the above conditions, then it is selected.
  - — If the "next unit" modifier is not set in the request, then the SLUN is used to locate the particular UQB being requested.
    - — If none is found, then GET_UNIT_STATUS returns an End message with an OFFLN status.
    - — If a matching UQB is found, then that UQB is selected.
- The following information is copied from the selected UQB into the buffer containing the GET UNIT STATUS request:
  - — MSCP unit number. (If the "next unit" modifier was set in the request, it is necessary to identify for which unit information is being returned.)
  - — Media ID.
  - — Disk geometry information.
    - o Number of sectors per track.
    - o Number of tracks per group.
    - o Number of groups per cylinder.
  - — Whether or not an RCT is present.
  - — Unit identifier.
- An MSCP status code is set up to be returned in the end message.

MSCP$K_ST_AVLBL	Unit is available.
MSCP$K_ST_OFFLN	Unit is offline to requesting host.
MSCP$K_ST_SUCC	Success (with MSCP$K_ST_ONLINE sub-code).

- GET_UNIT_STATUS then branches to SEND_END, which does the following:
  - — Stores the MSCP major status code in the buffer containing the GET UNIT STATUS request.
  - — Converts the received GET UNIT STATUS into an MSCP End message.
  - — Sends the End message to the requesting host.

5.7.2.1.4 SET_CONTROLLER_CHAR

Host settable characteristics are extracted from the command and inserted into the HQB corresponding to the controller specified in the command. An end message containing both host settable and non-host settable characteristics is then returned to the host which issued the SET CONTROLLER CHARACTERISTICS command.

- Stores the following information in the HQB corresponding to the remote host which issued the SET CONTROLLER CHARACTERISTICS command:
  - — Host settable controller flags from command (e.g. enable attention messages, enable miscellaneous error log messages, etc.).
  - — Local host's date and time. (Quadword date and time in command is not used.)

— Host access timeout from command.

NOTE

All values of host access timeout greater than 255 are replaced by 255. All values less than 10 are replaced by 10.

- Copies the following information from the DSRV into the buffer containing SET CONTROLLER CHARACTERISTICS command:
  - — Server's allocation class.
  - — Server's controller software version number.
  - — Server's controller flag settings.
  - — Server's controller timeout interval.
  - — Server's controller identifier.
- SET_CONTROLLER_CHAR then branches to SEND_END, passing to it the major status code MSCP$K_ST_SUCC. SEND_END does the following:
  - — Stores the MSCP major status code in the buffer containing the GET UNIT STATUS request.
  - — Converts the received SET CONTROLLER CHARACTERISTICS into an MSCP end message.
  - — Sends the end message to the requesting host.

5.7.3 Non-Sequential Non-Buffered Class Commands

The VMS based MSCP server deals with five nonsequential nonbuffered commands sent to it by a remote host:

Table 5–9: NonSequential NonBuffered Commands

Command	Definition
ACCESS	The purpose of this command is to verify that designated data can be read without error. Data is to be read from a unit, checked for any errors, and then discarded.
COMPARE CONTROLLER DATA	The controller is requested to do a consistency check of data from members of a shadow set virtual unit.
ERASE	All data in a specified region of of a unit is overwritten by zeros.
FLUSH	The controller is to flush cached commands or data from a host.
REPLACE	A logical block is marked as being replaced by a replacement block.

5.7.3.1 Access Command

The ACCESS command is not supported by the VMS based MSCP server and returns an end message bearing the "invalid command" status MSCP$K_ST_ICMD.

5.7.3.2 Replace Command

The VMS based MSCP server handles the REPLACE command in the same way as the ACCESS command, but for a very good reason. The local host is emulating an "MSCP speaking" controller which handles its own bad block replacement. Remote hosts should not be telling it which logical blocks to replace with replacement blocks.

5.7.3.3 Compare Controller Data and Flush Commands

Controllers that do not support caching must treat COMPARE CONTROLLER DATA and FLUSH commands as NOPs that always succeed. The VMS based MSCP server does nothing more than return an end message with the status code MSCP$K_ST_SUCC.

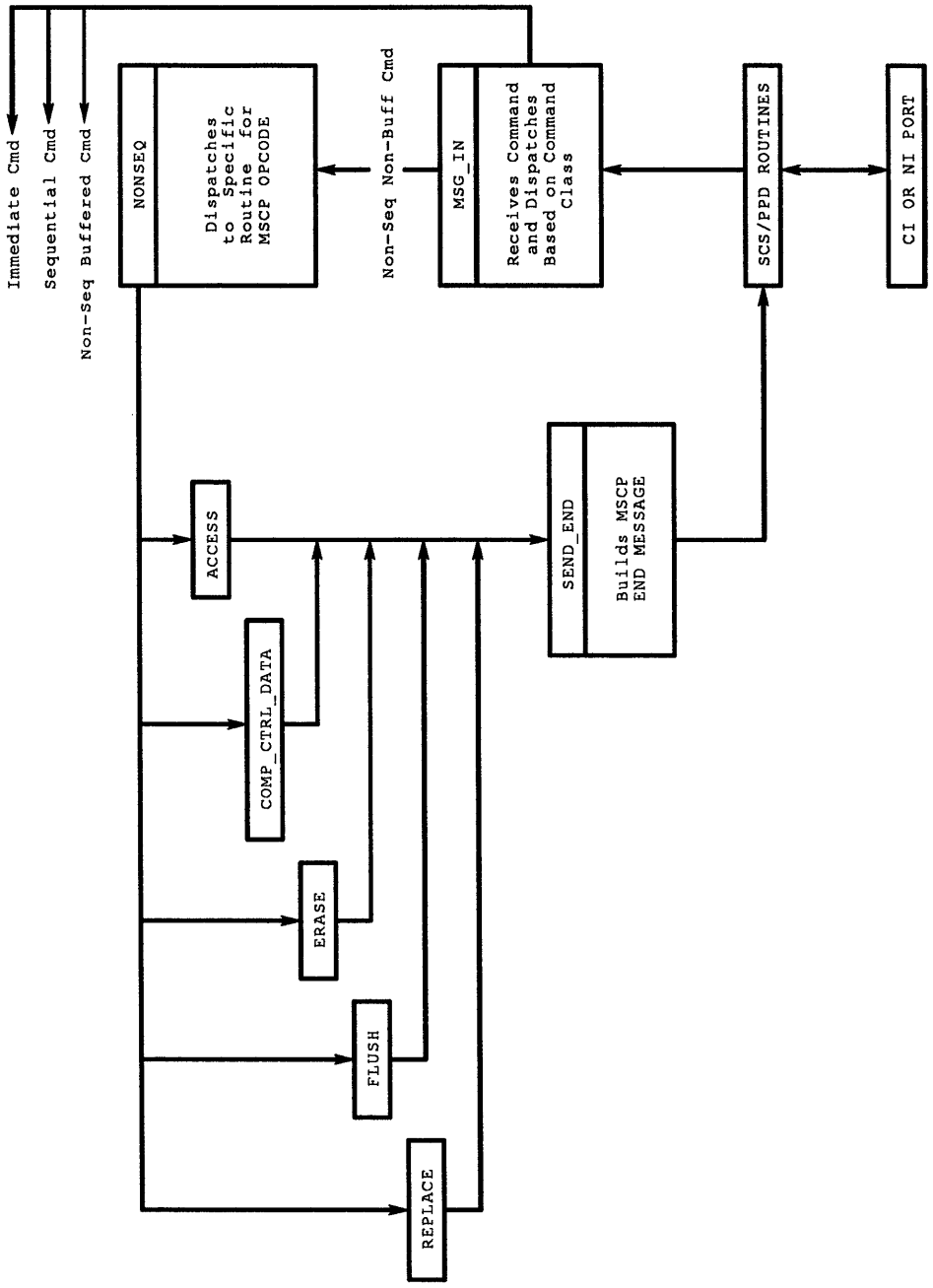
5.7.3.4 Erase Command

To handle the ERASE command, the server first verifies that the unit is not write protected. If the local controller for the unit is DSA based, it issues an *IO$_DSE* (data security erase) function to DUDRIVER for the unit. If the local controller is not DSA based, it must explicitly write zeros to the region specified in the ERASE command by issuing an IO$_WRITEPBLK function to the unit's driver.

If the server receives a nonsequential nonbuffered command for a local disk which has the online bit clear in its Unit Queue Block, an *MSCP$K_ST_AVLBL* status is returned in the End message.

Figure 5–18 illustrates the basic flow for NonSequential NonBuffered Commands.

Figure 5–18: NonSequential NonBuffered Command Flow



CXN-0005-14

5.7.4 Routines for Handling Non-Sequential Non-Buffered Commands

The following are outlines of the routines for handling nonsequential nonbuffered commands received by the VMS based MSCP server.

Routine NONSEQ passes to ACCESS, COMP_CTRL_DATA, and REPLACE the addresses of the MSCP command and the HRB corresponding to the remote host which issued the command. In addition to the MSCP command and HRB addresses, NONSEQ also passes to ERASE and FLUSH the address of the UQB corresponding to the unit referenced by the command.

5.7.4.1 ACCESS Routine

This command is not supported by the VMS based MSCP server. Routine ACCESS merely branches to PACKET_ERROR. PACKET_ERROR then branches to SEND_PKT to send to the remote host an end message bearing the status code MSCP$K_ST_ICMD (invalid command).

5.7.4.2 COMP_CTRL_DATA Routine

This command is treated as a NOP that always succeeds. Routine COMP_CTRL_DATA merely branches to SEND_END to send to the remote host an end message bearing the status code MSCP$K_ST_SUCC (success).

5.7.4.3 ERASE Routine

Overwrites with zeros a region on a disk unit, as specified in an MSCP ERASE command.

- If any of three conditions is present, then routine ERASE really has no work to do for the command. It merely branches to SEND_END to issue an end message to the remote host bearing an appropriate MSCP status code.
  - — If either the software write protect flag (UF_WRTPS) or the hardware write protect flag (UF_WRTPH) is set in the UQB, then no work can be done. So the MSCP status code sent in the end message is *MSCP$K_ST_WRTPR*.
  - — If the the byte count field in the ERASE command contains a 0, then this is a "zero byte transfer"; so there is no work to be done. In this case, the appropriate status code is "success", MSCP$K_ST_SUCC.
  - — If the operation exceeds the highest LBN on the unit, then the command is invalid. The status code MSCP$K_ST_ICMD is sent in this case.
- If none of the three conditions just described is present, then the IRP linked to the HRB (offset HRB$L_IRP_CDRP contains the address of this IRP) is used to pass the ERASE request to the the driver for the unit.

How the IRP is setup and used depends on whether the unit is on a local DSA controller, or a local non-DSA controller. The DEVCHAR2 field in the UCB for the unit is examined to see if the the MSCP flag is set. If the flag is set, the controller is DSA; if the flag is not set, then the controller is not DSA.

— If the controller is DSA, then the "data security erase" function, IO$_DSE, is supported.
  - o The IRP's function field is set to contain IO$_DSE.
  - o The IRP's starting LBN, offset, and byte count fields are set to reflect the entire part of the unit to be zeroed. (Since the IO$_DSE function is supported for disks on "MSCP speaking" controllers, there will be no need to perform this operation in "segments".)
  - o Routine DO_DISK is called to pass the IRP to DUDRIVER.
  - o If the operation completes successfully, then and end message is sent to the remote host. The end message contains the status code MSCP$K_ST_SUCC, indicating success, and the number of bytes set to zero.

    If the operation fails, then routine ERASE branches to XFER_ERR. At XFER_ERR, the VMS condition value returned by DUDRIVER is converted to an equivalent MSCP status code; and this status code is sent to the remote host in an end message.

— If the controller is not DSA, then the erase function is not supported and must be emulated.
  - o The erase function is emulated by setting up the IRP to actually write zeros to the specified region of the disk by means of an IO$_WRITEPBLK function.
  - o If the erase request involves more than a 127 blocks, then the operation will be "segmented".
  - o As described above in the DSA case, DO_DISK is called upon to actually pass the IRP to the driver for the non-DSA unit. If the the request is segmented, then this is done repeatedly until either the operation completes, or a failure occurs.
  - o If the operation completes successfully, then an end message is sent to the remote host. The end message contains the status code MSCP$K_ST_SUCC, indicating success, and the number of bytes set to zero.

    If the operation fails, then routine ERASE branches to XFER_ERR. At XFER_ERR, the VMS condition value returned by DUDRIVER is converted to an equivalent MSCP status code; and this status code is sent to the remote host in an end message.

NOTE

Routine XFER_ERR and a table equating VMS condition values with MSCP status codes are presented in a later section of this chapter covering sequential commands.

5.7.4.4 FLUSH Routine

This command is treated as a NOP that always succeeds. Routine FLUSH merely branches to SEND_END to send to the remote host an end message bearing the status code MSCP$K_ST_SUCC (success).

5.7.4.5 REPLACE Routine

This command is not supported by the VMS based MSCP server. Routine REPLACE merely branches to PACKET_ERROR. PACKET_ERROR then branches to SEND_PKT to send to the remote host an end message bearing the status code MSCP$K_ST_ICMD (invalid command).

5.7.5 Sequential Class Commands

The VMS based MSCP server supports four sequential class commands sent to it by a host:

Table 5–10: Supported Sequential Class Commands

Command	Purpose
AVAILABLE	to place a unit in the "available" state
ONLINE	to set the unit online to the host
SET UNIT CHARACTERISTICS	to set the unit's characteristics from the host's point of view
DETERMINE ACCESS PATHS	to have the unit identify itself to a secondary controller to which it is currently ported

All sequential class commands are passed to routine *SEQUENTIAL* by the MSCP server's SCS message input routine MSG_IN. If there is no Sequential Command active for this unit, no requests currently being processed, and no requests on the blocked queue, the command is processed. Routine SEQUENTIAL will dispatch to the appropriate routine. If there is no sequential command active for this unit and there are requests currently being processed for the unit, the command is placed at the tail of the Blocked queue. If there is a sequential command already in progress for this unit, the command is inserted at the tail of the Blocked queue.

Here is a summary of each of the four command specific routines:

- AVAILABLE

  An AVAILABLE command is issued to the controller by a host processing a dismount request for a unit on that controller.

  The UQB for each unit being served by the VMS based MSCP server contains a bitmap of client hosts. If the unit is online to a client host, then the bit in that bitmap corresponding to the client host is a "1"; otherwise, the bit is a "0".

First, the bit corresponding to the client host which sent the AVAILABLE command is cleared. If the bitmap also indicates that the unit is no longer online to any other client host, then

— The AVAILABLE flag will be set in the UQB.

— The unit will be unloaded if the "online count" field of the UCB indicates no host (not even the local host) has the unit online and the *Spindown* modifier was set in the AVAILABLE command.

- ONLINE

The MSCP ONLINE command is issued by a host as part of a request to mount a unit.

The MSCP server issues an IO$_PACKACK function to determine if the unit is reachable. A DSA device receiving this function will return a "success"; but a non-DSA device will return an "illegal I/O" status. Either is acceptable since the server merely wishes to see if it can reach the unit.

If the unit is reachable:

— The ONLINE flag is set in the unit's UQB.

— The bit corresponding to the remote host is set in the UQB's client ONLINE bitmap. (This indicates that the unit is now online to the remote host that issued the ONLINE command.)

— The UCB's count of hosts having the unit online is incremented.

— The HULB structure is initialized for the Host/Unit combination

— Various unit characteristics are returned to the remote host in an end message. These include the number of LBNs, volume serial number, multi-unit code, unit flags, unit identifier, and media ID.

If the unit is not reachable, then either an "offline" or "drive error" status code will be sent to the remote host in the end message.

- SET_UNIT_CHR

The SET UNIT CHARACTERISTICS command is used to control host settable unit characteristics, and to obtain unit characteristic information necessary to a host's disk class driver.

The only characteristic in this command which is of interest to the VMS based MSCP server is the *Enable Set Write Protect* modifier, MD_STWRP, in the MODIFIERS field of the command. The purpose of this flag, if set, is to allow the *Software Write Protect* unit flag, UF_WRTPS, to be host settable. The MD_STWRP modifier being set to "1" in the SET UNIT CHARACTERISTICS command does not in itself alter the current setting of the UF_WRTPS unit flag.

The VMS based MSCP server's interest in this flag is quite minimal. In fact, the only action taken by the server is to verify that the "software write protect" flag will be a zero in the end message returned to the remote host if the MD_STWRP modifier is a zero in this command.

The server doesn't even record the setting of the MD_STWRP modifier. An end message containing various unit characteristics, including the number of LBNs, volume serial number, multi-unit code, unit flags, unit identifier, and media ID is sent to the remote host.

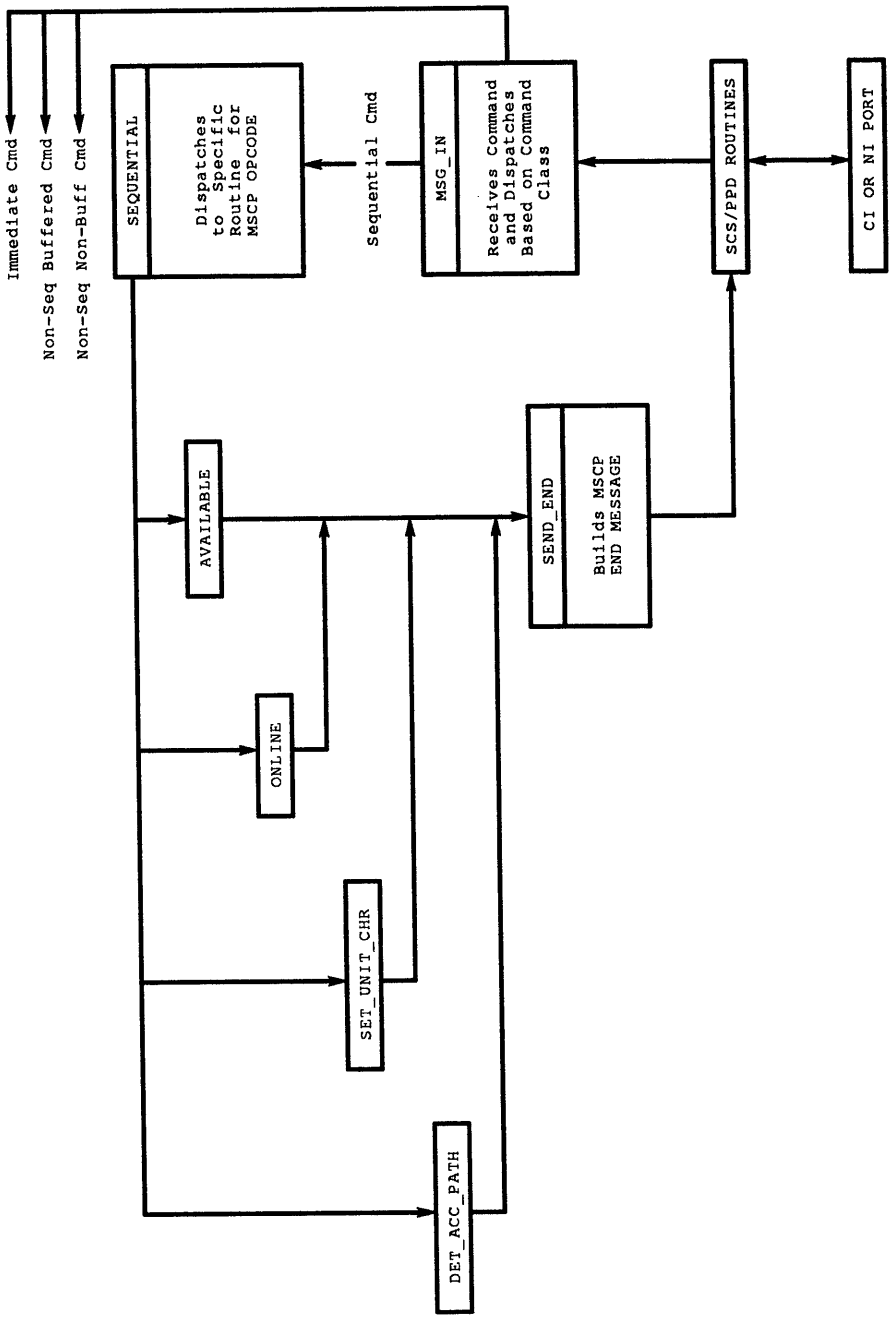
- DET_ACC_PATH

  The DETERMINE ACCESS PATHS command is used by the disk class driver to determine which controllers provide paths to a dual-pathed unit. When a DETERMINE ACCESS PATHS command is received, normally the unit would identify itself to any other controller to which it is ported, and then that other controller would send ACCESS PATH ATTENTION messages to hosts with which it is communicating. This is one way hosts would find out about an alternate path to the unit.

  However, the VMS based MSCP server does not provide any of this functionality. It effectively treats a DETERMINE ACCESS PATHS command as a NOP. It "pretends" it implements this command by merely sending an end message with a status code of MSCP$K_ST_SUCC (success) to the remote host which issued the command.

Figure 5–19 illustrates the general flow of Sequential Commands.

Figure 5–19: General Flow of Sequential Commands



5.7.6 Routines For Handling Sequential Commands

Following are detailed outlines of the four command specific routines for handling sequential commands.

Routine SEQUENTIAL passes to each of these routines the addresses of the the sequential MSCP command, the HRB corresponding to the host which sent the command, and the UQB corresponding to the unit referenced by the command.

5.7.6.1 AVAILABLE

Sets unit in AVAILABLE state for the client host which issued the AVAILABLE command, and in the AVAILABLE state for all client hosts if no other client host has the unit online.

- The HULB structure is located and deallocated
- UQB's ONLINE bitmap of client hosts is examined to determine whether or not the unit is online to the remote host which issued the AVAILABLE command.

  If the unit is online to the remote host, then

  — The bit in the ONLINE bitmap corresponding to the remote host is cleared so that it no longer is.

  — The address of the UCB for the unit is fetched from the UQB, and the online count field in the UCB (UCB$B_ONLCNT) is decremented.
- The UQB's ONLINE bitmap of client hosts is examined to determine if the unit is still being held online for any other client hosts.

  If not, then the following steps are performed:

  — The unit's state is set to "available for client hosts" by setting the AVAILABLE flag in the UQB.

  — If the UCB$B_ONLCNT field is zero, then the unit is no longer online to any host (not even to the local host). In this case, one of two additional steps will also be taken:

    o If the "spindown" modifier was not set in the AVAILABLE command, then an IO$_AVAILABLE function is issued to the driver for the unit.

    o If the "spindown" modifier was set, then an IO$_UNLOAD function is issued to the driver for the unit.

  If the driver for the unit returns to AVAILABLE a VMS condition value indicating that the IO$_AVAILABLE or IO$_UNLOAD function failed, then AVAILABLE does not continue as described here, but rather, branches to XFER_ERR to do the following:

    o Converts the VMS condition value into an equivalent MSCP status code. (A table of equivalences is provided at the end of this section.)

    o If the condition value was MEDOFL, VOLINV, or TIMEOUT, then the AVAILABLE flag is set in the UQB's STATE field; if the condition value was WRITLCK, then the WRTPH flag is set in the UQB's STATE field.

    o A branch is taken to SEND_END to issue an end message containing the equivalent MSCP status code.

  If the IO$_AVAILABLE or IO$_UNLOAD is successful, the following is performed by routine AVAILABLE.

— The hardware and software write protect flags in the UQB (UF_WRTPH and UF_WRTPS, respectively) are cleared.

- AVAILABLE now branches to routine SEND_END, passing it the major status code MSCP$K_ST_SUCC. SEND_END does the following:
  — Stores the MSCP major status code in the buffer containing the GET UNIT STATUS request.
  — Converts the received AVAILABLE into an MSCP end message.
  — Sends the end message to the requesting host.

  The following table lists the VMS condition values and the equivalent MSCP status codes into which they are converted by routine XFER_ERR:

Table 5–11: VMS Error Status to MSPC Status Translation

VMS Error	MSCP Status
SS$_ABORT	MSCP$K_ST_ABRTD
SS$_MEDOFL	MSCP$K_ST_AVLBL
SS$_VOLINV	MSCP$K_ST_AVLBL
SS$_WRITLCK	MSCP$K_ST_WRTPR
SS$_DATACHECK	MSCP$K_ST_COMP
SS$_CTRLERR	MSCP$K_ST_CNTLR
SS$_FORMAT	MSCP$K_ST_MFMTE
SS$_FORCEDERROR	MSCP$K_ST_DATA
SS$_PARITY	MSCP$K_ST_DATA (subcode = 1)
SS$_IVBUFLEN	MSCP$K_ST_HSTBF
SS$_TIMEOUT	MSCP$K_ST_OFFLN (subcode = MSCP$K_SC_UNKNO)

5.7.6.2 ONLINE

Sets a unit online to a client host if the unit is reachable by the VMS based MSCP server.

- If the ONLINE command was issued for either a shadow set master unit or a shadow set member, then no further processing is done for the command by this routine. Instead, an end message is sent to the host which issued the ONLINE command, this end message will contain the status code MSCP$K_ST_ICMD (invalid command).
- If the Online if for a Shadow Set Virtual Unit, verify that the Shadow set is still intact.
- An IO$_PACKACK function is issued for the unit to determine if it is reachable.

  A DSA device receiving a PACKACK returns a "success" status. But non-DSA devices cannot handle PACKACKs; so they return an "illegal I/O" status. Either status is acceptable since the MSCP server is merely trying to see if the unit is reachable.
- If the IO$_PACKACK returns a condition value of either SS$_NORMAL or SS$_ILLIOFUNC, then:
  — The ONLINE flag is set in the UQB$W_STATE field.

— Create the HULB structure for this Host/Unit combination and initialize
— If the UQB's ONLINE bitmap of client hosts indicates that the unit was not already online for the host which sent the ONLINE command, then:
  o Routine ONLINE sets the bit in the bitmap corresponding to the client host which sent the ONLINE command.
  o It increments the UCB$B_ONLCNT field if not a Shadow Set Virtual Unit.
— A branch is taken to COPY_CHAR to copy device characteristics into the buffer containing the ONLINE command.
  o Content of UCB$L_MAXBLOCK field (number of LBNs).
  o Volume serial number (if already mounted by local host or some other client host), or recognizable "bogus" value of ^X1234.
  o Multi-unit code.
  o Unit flags.
  o Unit identifier.
  o Media ID.

  COPY_CHAR then branches to SEND_END to convert the ONLINE command into an end message, and send the end message to the host which issued the ONLINE command.

- If the IO$_PACKACK function returned neither SS$_NORMAL nor SS$_ILLIOFUNC, then the device is not reachable via this server. In this case, an end message is sent to the remote host with one of two MSCP major status codes:
  — MSCP$K_ST_OFFLN (offline) if the condition value was wither SS$_MEDOFL or SS$_DEVNOTSHR.
  — MSCP$K_ST_DRIVE (drive error) otherwise.

5.7.6.3 SET_UNIT_CHR

Intended to control host settable unit characteristics, this command is treated almost as a NOP by the VMS based MSCP server.

- If the host which issued the SET UNIT CHARACTERISTICS is not among those client hosts indicated by the UQB's client bitmap as having the unit online, then an end message is sent to the host with the status MSCP$K_ST_ICMD (invalid command).
- If the MD_STWRP modifier (enable set write protect) is not set in the MODIFIERS field of the command, then the SET_UNIT_CHR clears the "software write protect" flag, UF_WRTPS, in the UNT_FLGS field of the command.
- Routine SET_UNIT_CHR falls into COPY_CHAR, which copies device characteristics into the buffer containing the command.
  — Content of UCB$L_MAXBLOCK field (number of logical blocks in host area of unit).
  — Volume serial number (if already mounted by local host or some other client host), or recognizable "bogus" value of ^X1234.
  — Multi-unit code.
  — Unit flags.

— Unit identifier.

— Media ID.

COPY_CHAR then branches to SEND_END to turn the command into an end message, and send the end message to the host which issued the command.

5.7.6.4 DET_ACC_PATH

For a VMS based MSCP server, this command is treated as a NOP. Nothing is done other than returning an end message with an MSCP$K_ST_SUCC (success) status code to the host which issued the command.

5.7.7 Sequential Commands, Nonsequential Commands, and Blocking

Sequential commands typically alter the context of a unit relative to a remote host through such actions as setting the unit AVAILABLE or ONLINE to that host. A sequential command must form a barrier around or through which a nonsequential command may not pass. All nonsequential commands received on the same SCS connection and for the same unit as a sequential command, but before the sequential command, must be completed before the sequential command is executed.

All nonsequential commands received on the same SCS connection and for the same unit as a sequential command, but after the sequential command, must be blocked until the sequential command is completed. Furthermore, multiple sequential commands received on the same connection for the same unit must be executed in the order in which they are received.

There are four components in the VMS based MSCP server's database that enforce these rules:

- UQB$W_CURRENT field in each UQB.

  This field contains a count of the number of MSCP commands received from remote hosts and that are currently active for the unit represented by the UQB.

  When a command specific to a particular unit is received by the server, it calls routine FIND_UQB to locate the UQB corresponding to the unit specified in the command. When this UQB is found, its CURRENT field is incremented.

  When routine SEND_END transmits the End message for this command back to the remote host, the UQB's CURRENT field is decremented after resources used in processing the command are released and the command's HRB is deallocated.

- UQB$V_SEQ bit in each UQB's FLAGS field.

  When set, this flag indicates that a sequential command is currently being executed. When clear, this flag indicates there is currently no sequential command being executed.

  If a sequential command for a unit has been received but is waiting for other commands received before it to complete, this flag is clear. It is not set until a sequential command actually begins execution.

- UQB's queue of blocked commands.

  In this queue reside nonsequential HRBs waiting for sequential commands to complete, and sequential HRBs waiting for nonsequential commands to complete.

The FLINK and BLINK for the head of this queue are at offsets *UQB$L_BLOCKED_FL* and *UQB$L_BLOCKED_BL*, respectively.

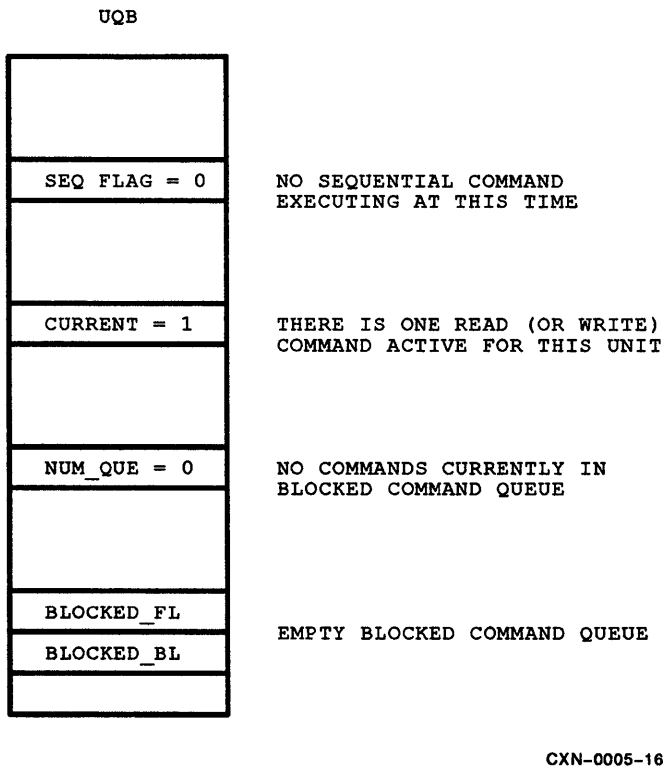
- UQB$W_NUM_QUE field in each UQB.

  This field contains the count of the number of HRBs in the UQB's queue of blocked commands.

5.7.7.1 Basic Scenario

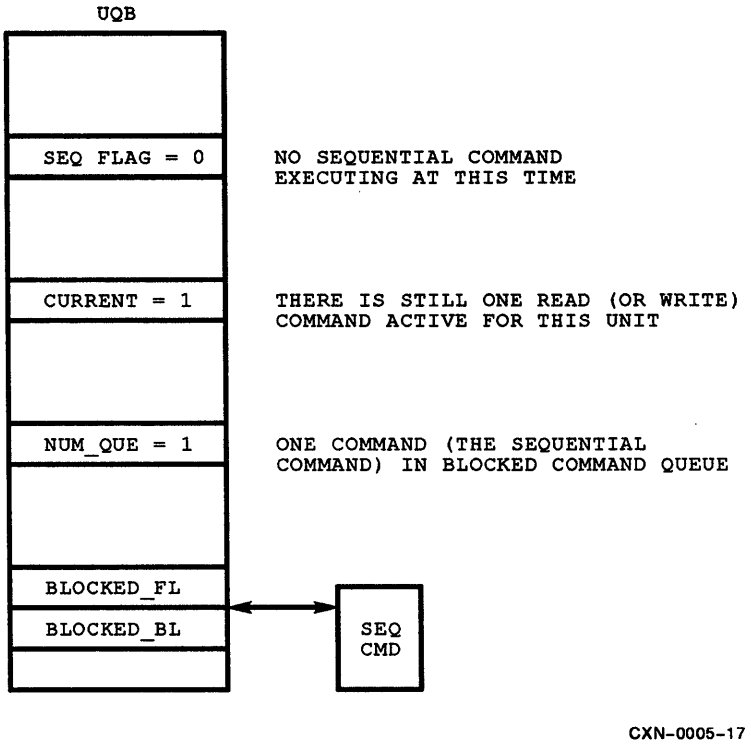
To understand the blocking and unblocking mechanism used by the MSCP server, begin by assuming that it is receiving and processing nonsequential commands (e.g. READs and WRITEs) for some unit. The UQB$W_CURRENT field is nonzero; it contains the number of these currently active commands. The UQB$V_SEQ flag is clear since there is no sequential command in execution. The queue of blocked commands is empty, and the NUM_QUE field is presently 0. Figure 5–20 illustrates this condition.

Figure 5–20: Processing NonSequential Commands with No Sequential Commands Issued



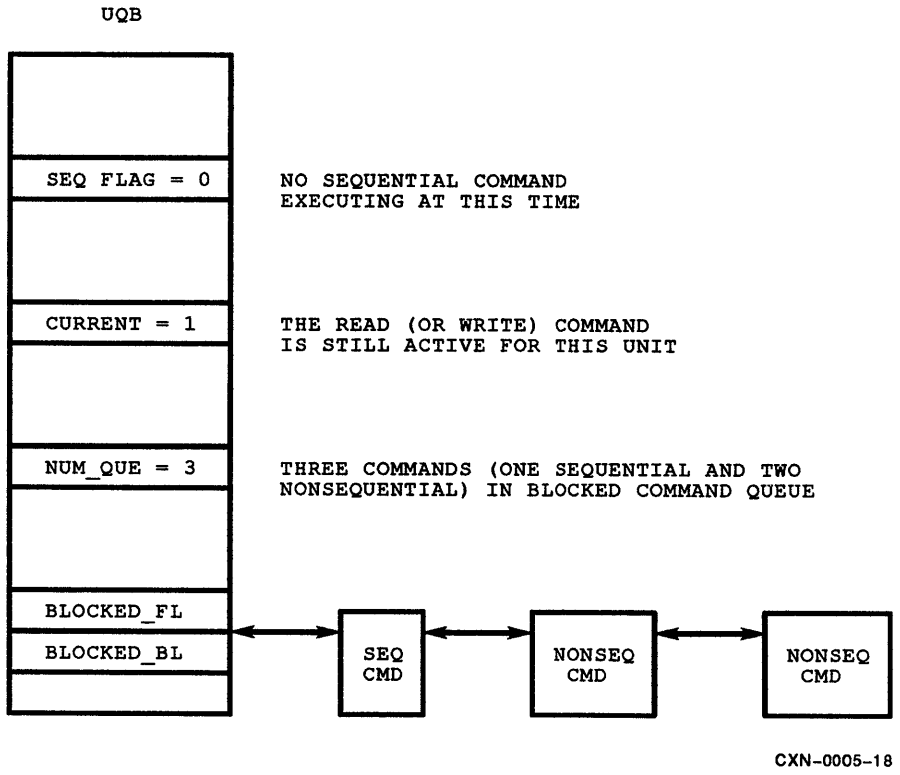
Then a sequential command is received. Since the CURRENT field contains at least a 1, FIND_UQB will increment it to at least a 2 when it finds the UQB for the unit to which the sequential command is directed. The CURRENT field now being greater than 1 indicates that at least one other command is already in execution. The sequential command's HRB is inserted into the blocked command queue, the NUM_QUE field is incremented to 1, and the CURRENT field is decremented since the sequential command is blocked and will not be put into execution yet. Figure 5–21 illustrates this condition.

Figure 5–21: Sequential Command Received While Processing NonSequential Commands



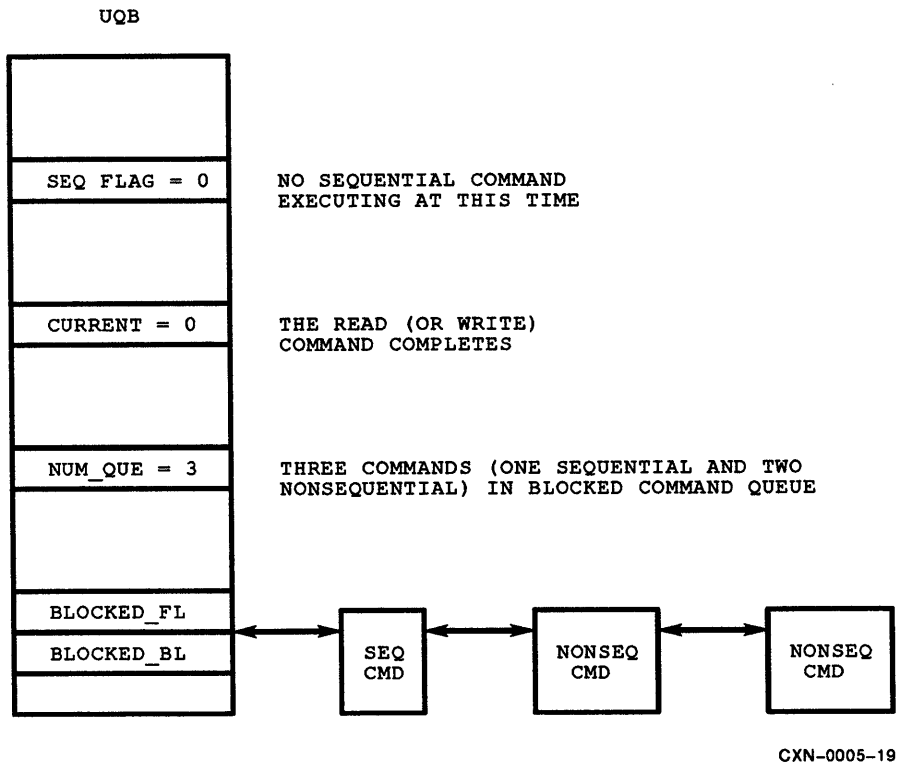
As explained earlier in this chapter, routines NONSEQ and NONSEQB dispatch nonsequential commands to their specific handlers. Before doing so, these routines examine the UQB$V_SEQ flag and find it still clear. But then they examine the NUM_QUE field and find it nonzero. Instead of dispatching nonsequential commands, they insert them into the blocked command queue behind the sequential command and increment NUM_QUE. Figure 5–22 illustrates this condition.

Figure 5–22: Sequential Command Pending With NonSequential Commands Arriving



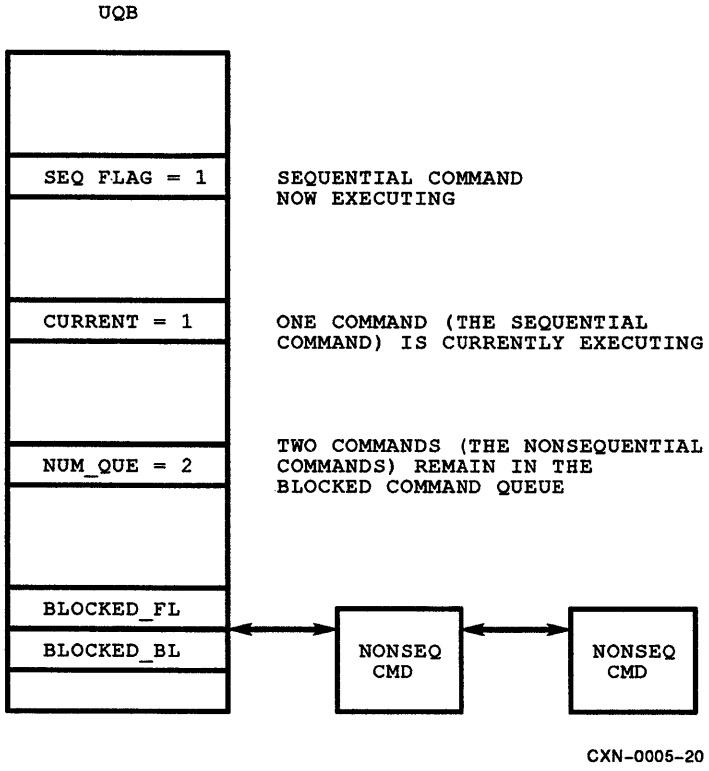
Active nonsequential commands received prior to the sequential command are allowed to complete. As each one does, routine SEND_END transmits the end message, and then calls routine UNBLOCK which decrements the CURRENT field. Figure 5–23 illustrates this condition.

Figure 5–23: Currently Executing Commands Have Completed With Commands Queued



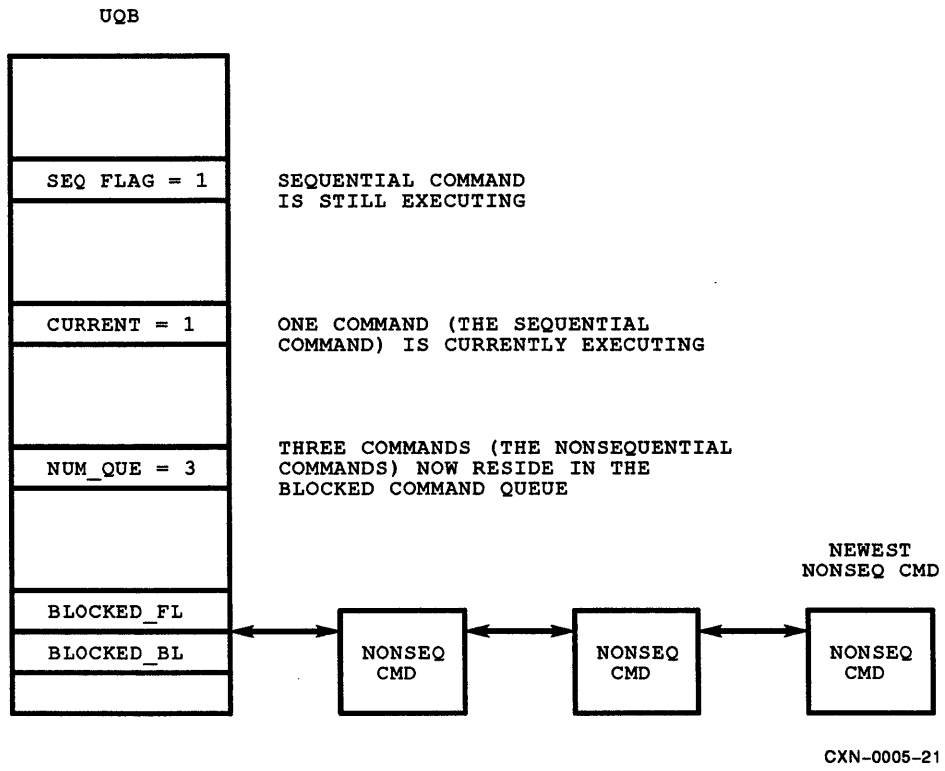
When the CURRENT field goes to zero, the sequential command is removed from the blocked command queue and placed in execution. As this is done, the NUM_QUE field is decremented, the CURRENT field is incremented to 1, and the UQB$V_SEQ flag is set. Figure 5–24 illustrates this condition.

Figure 5–24: Sequential Command Begins Execution



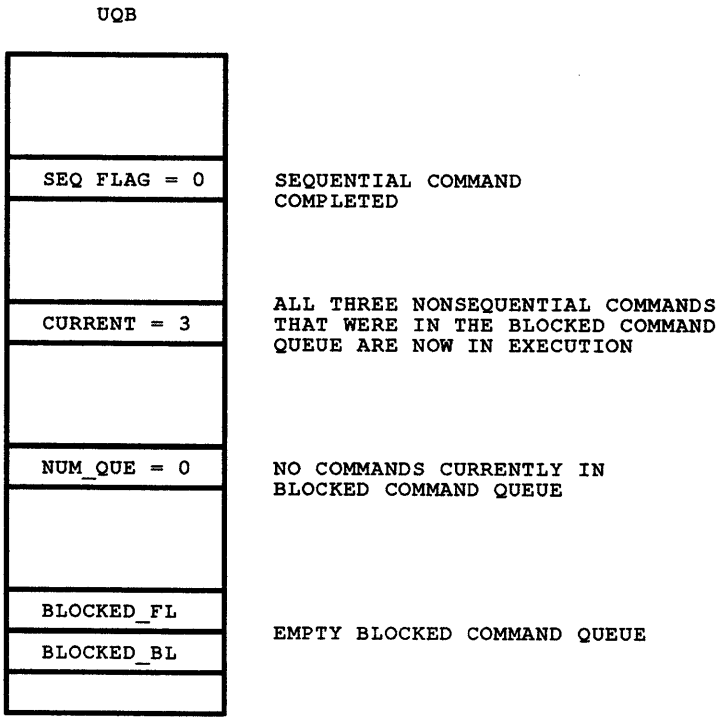
While the sequential command is executing, the UQB$V_SEQ flag being set causes NONSEQ and NONSEQB to continue inserting nonsequential commands into the blocked command queue. (This is important since, if there are no nonsequential commands behind the sequential command in the blocked command queue at the moment that the sequential command begins execution, then the NUM_QUE field will go to 0. However, while the sequential command is executing, it is necessary that newly received nonsequential commands still be inserted into the blocked command queue.) Figure 5–25 illustrates this condition.

Figure 5–25: Sequential Command Executing with NonSequential Commands Arriving



Finally, the sequential command completes execution. SEND_END transmits the End message for the sequential command, and then clears the UQB$V_SEQ as part of the procedure for deallocating the sequential command's HRB. Then SEND_END calls UNBLOCK. UNBLOCK decrements the CURRENT field to 0; then it loops, resuming each of the commands in the blocked command queue, incrementing the CURRENT field and decrementing the NUM_QUE field for each command resumed. Figure 5–26 illustrates this condition.

Figure 5–26: Sequential Command Completes and NonSequential Commands Resume



Until the last HRB in the blocked command queue is resumed, the NUM_QUE field remains nonzero. All subsequent commands received for this unit will be inserted into the queue, and then resumed by UNBLOCK when they reach the head of the queue.

NOTE

Even though a GET UNIT STATUS command is an immediate class command, it does increment the CURRENT field, so it will cause a sequential command to wait. However, the converse is not true; a GET UNIT STATUS command will not block, even if a sequential command is executing.

5.7.7.2 Special Case

While processing any sequential or nonsequential command, the CURRENT field is always at least 1. A subsequent sequential command will be inserted at the tail of the blocked command queue. As UNBLOCK resumes nonsequential HRBs in the blocked command queue, the sequential HRB moves toward the head of the queue.

Eventually, UNBLOCK encounters the sequential HRB. When this happens, UNBLOCK leaves the sequential command in the blocked command queue and exits the loop. As each resumed nonsequential command completes, UNBLOCK is called to decrement the CURRENT field. When the CURRENT field again goes to zero, the sequential command at the head of the queue is executed, and events progress as described in the previous section.

5.8 Error Handling

There is very little error recovery built into the VMS based MSCP server because there are very few situations it needs to deal with.

If a remote host sends the server an MSCP command requesting information, the server merely retrieves the information from its own database and sends that information to the host in an End message. Such requests don't really give rise to situations which could lead to error recovery by the server.

If a remote host sends the server an MSCP command which involves transferring data or altering the context of a unit, then an IRP is constructed and passed to the appropriate driver to actually accomplish the task. If an error is going to occur, it will most likely occur within the driver. Such errors will merely be translated into equivalent MSCP status codes by calling routine XFER_ERR, and then be sent to the remote host in an End message.

There are, however, two situations wherein the server plays an active role in dealing with errors:

- Loss of SCS Connection with a Remote Host.

  The remote host, if it is still running, will attempt to resynchronize its activity with the local server. The basic philosophy behind the server's handling of such a situation is to guarantee that, from the server's point of view, there are no commands active from the remote host. It does this by scanning the queue of HRBs attached to the remote host's HQB, marking each HRB as being "aborted". All HRBs that are not currently acquiring mapping resources or being processed by a driver are deallocated. HRBs that are within the "jurisdiction" of mapping routines or a driver will be deallocated when they return for further processing by the server; this is because they will then be seen as having been marked as "aborted". When the last HRB in the HQB's queue is deallocated, the HQB will be deallocated.

  The remote host resynchronizes with the server by establishing a new SCS connection. The server will allocate a new HQB for the remote host as part of its accepting the CONNECT request. The remote host can then resubmit to it, one at a time, the MSCP commands that were active at the time the original connection was lost. This, of course, presumes that the remote host has not rebooted, and thus still has commands outstanding for this server.
- BUGCHECKs

  There are various situations which lead to the MSCPSERV BUGCHECK, all of which are fatal.

When the server accepts a CONNECT request from a remote disk class driver, it declares to the SCS layer that its routine VC_ERR will handle any loss of that SCS connection. Control is passed to VC_ERR when a failure occurs in the virtual circuit between the local host and the remote host, or when the remote disk class driver issues a DISCONNECT.

- VC_ERR begins by fetching from the *CDT$L_AUXSTRUC* field of the CDT the address of the HQB for the remote host at the other end of the lost connection.
- Routines *SCAN_RSPID_WAIT, SCAN_RDT* and *SCAN_MSGBUF_WAIT* are called to unhook all CDRPs for the CDT on the failed connection.
- Then it loops through the queue of HRBs attached to the HQB to dequeue all nonactive requests.
  - Each HRB (and hence MSCP command from the remote host) is marked as aborted by setting the ABORT bit in the HRB$W_FLAGS field.

    NOTE

    This ABORT flag should not be confused with the ABORTWS flag set in an HRB when processing the MSCP ABORT command. The ABORT flag is used to signify that a command is being aborted due to the failure of the SCS connection with the remote host which sent the command. No status will be returned to that remote host since there is no connection with its disk class driver.

    The ABORTWS is used to signify that a command has been aborted at the explicit request of a remote host. It is "aborted with status" since an MSCP$K_ST_ABORTD status code is sent to the host.

  - If the HRB's state is neither HRB$K_ST_DRV_WAIT nor HRB$K_ST_MAP_WAIT, then its CDRP is within the jurisdiction of the MSCP server; so its CDRP is removed from any wait queue in which it currently resides.

    CDRPs for HRBs in the DRV_WAIT and MAP_WAIT state are left where they are for now. Such CDRPs have already been handed to a driver for processing, or are currently being allocated SCS mapping resources. Therefore, they are currently outside the "jurisdiction" of the MSCP server.
- Again, VC_ERR loops through the queue of HRBs attached to the HQB. This time it releases the SCS message buffer held by the HRB, if it has one.
- A formal SCS DISCONNECT is done to clean up the SCS database for the lost connection, and mainly to release the CDT for the connection if the *HQB$V_PATHMOVE* flag is not set.
- If the connection was broken to facilitate a Load Balance (HQB$V_PATHMOVE is set), the DISCONNECT is issued with an *SCS$C_USE_ALTERNATE_PORT* status.
- The client host ONLINE bitmap of each UQB queued to the DSRV is examined to see if the unit is online to the remote host at the other end of the failed SCS connection. If so, the MSCP server issues an AVAILABLE command to an alternate entry point into its own sequential command handler.

  The primary effect of this is the clearing of the bit in the UQB's ONLINE bitmap corresponding to the remote host; so the unit is no longer online to that host. (See the previous section in this chapter covering the AVAILABLE command for additional details.)

To do this, an HRB is allocated along with an attached message buffer. The AVAILABLE command is built in the message buffer, the VCFAILED flag is set in this HRB's FLAGS field, and the HRB is queued to the HQB.

- HRBs not in the DRV_WAIT or MAP_WAIT states and that do not have their VCFAILED flag set are deallocated. Also, any resources (such as transfer buffers, IRPs, etc.) are released.
- Finally, the HQB for the remote host itself is deallocated if no HRBs remain queued to it.

NOTE

HRBs that may still be queued to it are those that were found to be in the DRV_WAIT or MAP_WAIT states earlier, or that have their VCFAILED flag set. When either the driver or mapping routines return them to the jurisdiction of the server, the ABORTED flag set will be seen to be a "1" in the HRB. This will cause the HRB to be passed to routine CLEANUP_HRB where it will be deallocated and its resources released.

CLEANUP_HRB will also deallocate the HQB in this case as well. The AVAILABLE command issued by VC_ERR is a sequential class command. Consequently, it will wait unit all other commands ahead of it are completed. Then it completes. The VCFAILED flag being set in its HRB causes CLEANUP_ HRB to deallocate the HQB if there are no other HRBs queued to the HQB; this should be the case since all other commands completed before it.

BUGCHECKs within the VMS based MSCP server are always Fatal and are of the following types: MSCPSERV, DISKSERVE or DOUBLDEALO. They fall into five general categories:

- An immediate, sequential, or nonsequential nonbuffered MSCP command containing an "unexpected" opcode is received from a remote host.

  Table 5–12 through Table 5–14 list the "expected" opcodes for each of these three classes. As described earlier in the chapter, not all of these are implemented as in a normal DSA controller.

Table 5–12: Expected Opcodes for Immediate Commands

Opcode	Definition
MSCP$K_OP_ABORT	Abort a command.
MSCP$K_OP_GTCMD	Get command status.
MSCP$K_OP_GTUNT	Get unit status.
MSCP$K_OP_STCON	Set controller characteristics.

Table 5–13: Expected Opcodes for Sequential Commands

Opcode	Definition
MSCP$K_OP_AVAIL	Set unit available.
MSCP$K_OP_ONLIN	Set unit online.
MSCP$K_OP_STUNT	Set unit characteristics.
MSCP$K_OP_DTACP	Determine access paths.

Table 5–14: Expected Opcodes for NonSequential NonBuffered Commands

Opcode	Definition
MSCP$K_OP_ACCES	Access data.
MSCP$K_OP_CMPCD	Compare controller data.
MSCP$K_OP_ERASE	Erase data.
MSCP$K_OP_FLUSH	Flush host buffers.
MSCP$K_OP_REPLC	Replace data.

If an MSCP command is in one of these three classes, but the opcode within the message is not listed in the appropriate table above, then the MSCP server causes a *MSCPSERV BUGCHECK.*

- While handling the failure of an SCS connection with a remote host, or while handling an ABORT command from a remote host, an HRB is found whose STATE field contains an "unacceptable" value.

  In the connection failure case, the HRB can be any of those queued to the HQB associated with the remote host at the other end of the lost connection. In the ABORT command case, the HRB would be associated with the command to be aborted.

  Table 5–15 lists the "acceptable" request states in the HRB$W_STATE field for the connection failure case:

Table 5–15: Acceptable Request States for Connection Failures

State	Definition
HRB$K_MSG_WAIT	Waiting for SCS msg. buffer/send credit.
HRB$K_SEQ_WAIT	Waiting for completion of sequential cmd.
HRB$K_BUF_WAIT	Waiting for local MSCP transfer buffer.
HRB$K_SNDAT_WAIT	Waiting for completion of block data xfer.
HRB$K_DRV_WAIT	Waiting for driver to complete request.
HRB$K_MAP_WAIT	Waiting for SCS mapping resources.

Finding an HRB with an "unacceptable" STATE field in either of these cases will cause the MSCP server to MSCPSERV BUGCHECK.

- While processing an AVAILABLE, ERASE, READ, or WRITE, an IRP may be returned to the server by a driver indicating the unit is suspect: SS$_MEDOFL, SS$_DEVOFFLINE, SS$_VOLINV, or SS$_TIMEOUT. The server will decrement the online count field in the UCB for each remote host having the unit online. If the online count field becomes negative, the server will MSCPSERV BUGCHECK.
- When starting up the Load Monitoring thread, either there is insufficient nonpaged pool for the repeating timer queue entry or there is insufficient nonpaged pool for the Fork Block. The MSCP server will exit with a Fatal DISKSERVE BUGCHECK.
- When Deallocating buffers back to the local transfer buffer pool, the buffer is found to already be in the available pool. This will cause the MSCP server to exit with a Fatal DOUBLDEALO BUGCHECK.

Appendix A

Symbol Tables and Data Structures

There are generally four methods of accessing data structure definitions. They are as follows:

1. Using SDA's predefined symbol tables: (ie: *SDA> read sys$system:{table}.stb*)
2. Extracting from the Public Libraries: (ie: *sys$share:starlet.mlb, sys$share:lib.mlb*)
3. Assembling the source SDL files: (ie: *[sysloa.lis]cluster.sdl*)
4. Manually creating user defined symbol tables

The general naming convention of a structure definition is to begin the name of the macro that creates the structure with a *dollar sign,* followed by the *structure name,* followed by the word *DEF* for definition. An example of the name of the macro defining the Unit Control Block (UCB) structure follows:

$UCBDEF

A.1 SDA Symbol Tables

The VMS distribution kit includes several default symbol tables. These symbol table files contain a subset of the structure definitions and symbols defined in the public libraries. The current list of supplied symbol tables is as follows:

```
DCLDEF.STB        DECDTMDEF.STB     IMGDEF.STB        NETDEF.STB
REQSYSDEF.STB     RMSDEF.STB        SCSDEF.STB        SYS.STB
SYSDEF.STB
```

SYS.STB is automatically loaded by the Analyze/*{system,crash}* utility. It contains symbols for the bugcheck codes, several system locations, several control region locations, the sysgen parameter offsets, memory management locations, some of the scheduler global locations, as well as many others.

SYSDEF.STB is another very useful symbol table. It **must be read in manually** using the following syntax:

SDA> *read sys$system:sysdef.stb*

The following structures are defined in the SYSDEF symbol table:

```
$ACBDEF         $CXBDEF         $KFDDEF         $PCBDEF         $SPLDEF
$ACFDEF         $DDBDEF         $KFEDEF         $PFLDEF         $TQEDEF
$ACLDEF         $DDTDEF         $KFPBDEF        $PHDDEF         $TTYDEFS
$ADPDEF         $DPTDEF         $KFRHDEF        $PQBDEF         $UCBDEF
$AQBDEF         $FCBDEF         $LCKCTXDEF      $PRVDEF         $VADEF
$ARBDEF         $FKBDEF         $LDRIMGDEF      $PSLDEF         $VCADEF
$CCBDEF         $GSDDEF         $LKBDEF         $RPBDEF         $VCBDEF
$CEBDEF         $IDBDEF         $LNMSTRDEF      $RSBDEF         $VECDEF
$CHPCTLDEF      $IPLDEF         $LOGDEF         $RSNDEF         $WCBDEF
$CHPRETDEF      $IRPDEF         $MPBDEF         $RVTDEF
$CPUDEF         $IRPEDEF        $MTLDEF         $SECDEF
$CRBDEF         $JIBDEF         $ORBDEF         $SPLCODDEF
```

SCSDEF.STB should be read in for viewing VAXcluster data structures. It contains symbols for the following structures:

```
$CDDBDEF        $CLUICBDEF      $LILDEF         $PEMCOMPDEF     $UCBDEF
$CDLDEF         $CLUPBDEF       $PAERDEF        $PEMREGDEF      $UCBNIDEF
$CDRPDEF        $CLURCBDEF      $PAPDTDEF       $PPDDEF         $VCIBDEF
$CDTDEF         $CNCTDEF        $PAREGDEF       $RDDEF          $VCIBDLLDEF
$CIBDDEF        $CRBDEF         $PAUCBDEF       $RDTDEF         $VCRPDEF
$CIBDTDEF       $CSBDEF         $PBDEF          $RHDRDEF        $VCRPLANDEF
$CLSMSGDEF      $CXBDEF         $PDTDEF         $SBDEF          $BUSDEF
$CLUBDEF        $DCBEDEF        $PEERLDEF       $SCSDEF         $NISCADEF
$CLUBTXDEF      $DYNDEF         $PEMCHDEF       $SDIRDEF        $PORTQBDEF
$CLUDCBDEF      $EMBDEF         $PEMCLSTDEF     $TR_ERRDEF      $VCDEF
```

The provided symbol tables only contain a subset of the data structures utilized by VMS. If after reading in the STB files your structure is still undefined, you will need to continue your search.

A.2 Public Libraries

The VMS distribution kit includes two library files (*STARLET.MLB* and *LIB.MLB*). These libraries describe a majority of the VMS data structure definitions. The libraries are provided in two formats, an ASCII text file, (BLISS source code) with an extension of **.REQ** and the macro library itself with the extension **.MLB**.

Each entry in the BLISS source code of the libraries has the following format:

```
macro VCB$B_TYPE     = 10,0,8,0 %;      ! structure type of VCB
macro VCB$V_WRITE_SM = 11,1,1,0 %;      ! Storage map is write accessed
macro VCB$B_LRU_LIM  = 77,0,8,1 %;      ! VOLUME DIRECTORY LRU SIZE LIMIT
```

In the above example, the VCB indicates that these fields are part of the Volume Control Block structure. The letter following the dollar sign indicates the data type (see Section A.5 for a list of data types) and the name following the underscore indicates the unique field name.
The four comma separated fields following the equal sign represent the following:

- Byte offset into the structure (ie: 10,11,77)
  This field indicates at what byte within a structure the field is located.
- Starting bit offset within the field (ie: 0,1,0)
  This item indicates at which bit within the field the data begins. A value of zero is bit zero, A value of one is bit one, etc.

- Size of the field in bits (ie: 8,1,8)
  This field would indicate 1 bit for a bit field, 8 bits for a byte field, 32 bits for a longword, etc.
- Whether the data is signed or not (ie: 0,0,1)
  For this field a value of **one** indicates that this is a signed field. A value of **zero** indicates that this is an unsigned field.

To determine if a given structure is defined in either of these libraries, you can use the VMS search command to look through the REQ files, or use the librarian utility to list the contents of the MLB files. An example of looking for the DSRV structure (Disk Server Structure used with MSCP serving) follows:

Example using the SEARCH command to look for $dsrvdef

$search sys$share:lib.req module,dsrv/match=and
!* MODULE $DSRVDEF[1] ***

Example using the librarian to check the macro library, LIB

$library/list sys$share:lib.mlb

```
Directory of MACRO library SYS$COMMON:[SYSLIB]LIB.MLB;2 on 14-MAR-1992 17:15:33
Creation date:  12-MAY-1991 13:12:58      Creator:  VAX-11 Librarian V04-00
Revision date:  12-MAY-1991 13:13:14      Library format:   3.0
Number of modules:    587                 Max. key length:  31
Other entries:          0                 Preallocated index blocks:     75
Recoverable deleted blocks:      0        Total index blocks used:       27
Max. Number history records:      20      Library history records:        0

$$IO_ROUTINES_DATADEF
$$SYSTEM_PRIM_DATADEF
$$TTYDIALTYPDEF
$ABDDEF
$ACBDEF
$ACFDEF
$ACMDEF
$ADBDEF
$ADPDEF
...
$DSRVDEF
$DTSSDEF
$DYNDEF
$ECBDEF
...
```

Once you have determined that a given structure is defined in either of the libraries, you will need to **extract** that information into a *symbol table* that will be readable by SDA. To accomplish this, a small macro program can be written. An example dialogue with SDA and creating the symbol table follows:

[1] The dollar sign and the DEF are part of the actual name of the macro module.

```
$analyze/system
VAX/VMS System analyzer
SDA>!Read in predefined symbol tables
SDA> read sys$system:sysdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SYSDEF.STB;1
SDA> read sys$system:scsdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SCSDEF.STB;1
SDA>!
SDA>!Attempt to format the DSRV data structure
SDA> format @scs$gl_mscp
%SDA-E-NOSYMBOLS, no "DSRV" symbols found to format this block
SDA>!
SDA>!The dsrv structure is not defined in the provided symbol tables
SDA>!Check the public libraries...
SDA> spawn
$!
$ search sys$share:starlet.req module,dsrv/match=and
%SEARCH-I-NOMATCHES, no strings matched
$!
$ search sys$share:lib.req module,dsrv/match=and
!*** MODULE $DSRVDEF ***
$!
$! Found the structure definition in the LIB library. Write a short
$! MACRO program to extract the information into a symbol table
$ create dsrvdef.mar
.library /sys$share:lib.mlb/
                          ;Note that the word GLOBAL on the
                          ;     following line MUST BE CAPS
$dsrvdef GLOBAL           ;Extract the DSRV structure definitions
.end
^Z
$!
$ macro dsrvdef.mar
$ link/symbol_table/noexecutable dsrvdef.obj
%LINK-W-USRTFR, image NL:[].EXE; has no user transfer address
$ logout
  Process SYSTEM_1 logged out at 14-MAR-1992 17:46:16.14
SDA>!Read in the newly created symbol table
SDA> read dsrvdef.stb
%SDA-I-READSYM, reading symbol table  SYS$SYSROOT:[SYSMGR]DSRVDEF.STB;1
SDA> format @scs$gl_mscp
8053DDD0   DSRV$L_FLINK                  00002850
8053DDD4   DSRV$L_BLINK                  000007AC
8053DDD8   DSRV$W_SIZE                       076C
8053DDDA   DSRV$B_TYPE                     69
8053DDDB   DSRV$B_SUBTYPE                01
8053DDDC   DSRV$W_STATE                      0000
8053DDDE   DSRV$W_BUFWAIT                0000
...
```

A.3 SDL files

VMS engineering provides Structure Definition Language files as part of the results disk produced from a VMS build. The SDL language is a general data Structure Definition Language. A conversion utility exists to take the SDL generic form definition and create a language specific version of the information. Note that the SDL.EXE image and the language specific conversion modules are **NOT** typically **shipped with** the **VMS** distribution. To convert the SDL file to a macro language format, the following files must exist:

sys$system:SDL.EXE

sys$share:SDLMACRO.EXE

An example of using SDL to create a symbol table follows:

```
$ analyze/system

VAX/VMS System analyzer
SDA> read sys$system:sysdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SYSDEF.STB;1
SDA> read sys$system:scsdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SCSDEF.STB;1
SDA>!
SDA>!Check if the structure is described in the predefined symbol tables
SDA> format/type=cluqf 804F6FE0
%SDA-E-NOSYMBOLS, no "CLUQF" symbols found to format this block
SDA>!Not defined
SDA> spawn
$!
$! Check the public libraries
$ search sys$share:starlet.req module,cluqf/match=and
%SEARCH-I-NOMATCHES, no strings matched
$!
$ search sys$share:lib.req module,cluqf/match=and
%SEARCH-I-NOMATCHES, no strings matched
$!
$! Not in the public libraries, must build from the SDL file...
$ sdl/language=macro v55disk:[sysloa.lis]cluster.sdl
$!
$! You must now edit the resulting macro file to force the creation
$!  of GLOBAL symbols and to invoke the macros
$ edit/edt/nocommand cluster.mar
*s /EQU=<=>/EQU=<==>/ w
    5           .MACRO  $CLSMSGDEF,..EQU=<==>,..COL=<:>
  596           .MACRO  $CLMDRSDEF,..EQU=<==>,..COL=<:>
  625           .MACRO  $CNCTDEF,..EQU=<==>,..COL=<:>
  667           .MACRO  $CLUBTXDEF,..EQU=<==>,..COL=<:>
  702           .MACRO  $CLUQFDEF,..EQU=<==>,..COL=<:>
  764           .MACRO  $INCRNFDEF,..EQU=<==>,..COL=<:>
6 substitutions
*insert 99999
       $clsmsgdef
       $clmdrsdef
       $cnctdef
       $clubtxdef
       $cluqfdef
       $incrnfdef
       .end
       ^Z
*exit
SYS$SYSROOT:[SYSMGR]CLUSTER.MAR;2 841 lines

$ macro cluster.mar
$ link/symbol_table/noexecutable cluster.obj
%LINK-W-USRTFR, image NL:[].EXE; has no user transfer address
$ logout
  Process SYSTEM_1 logged out at 14-MAR-1992 18:11:28.53
SDA>!Read the newly defined structure into memory
SDA> read cluster.stb
%SDA-I-READSYM, reading symbol table  SYS$SYSROOT:[SYSMGR]CLUSTER.STB;1
SDA> format/type=cluqf 804F6FE0
804F6FE0   CLUQF$T_IDENT                    ""
804F6FE1                                    804F72
           CLUQF$C_ACT_LENGTH
```

```
804F6FE4                                        8050F950
804F6FE8                                        03650220
804F6FEC    CLUQF$W_VERSION
...
```

A.4 User Created Symbol Tables

The last possibility is that you are examining memory locations for which a VMS provided structure symbol table does not exist. In this instance, you will either have to use the system source code to determine what information is located at what offset in the structure each time you examine the information, or you can create your own structure definition. By examining the VMS source code, it is determined that during a lock manager detected exception, register R2 contains the address of the message portion of a lock request packet. It is determined that at negative offsets from the lock message itself we can find the SCS header information and the PPD header information. An example of looking at a Lock Message structure follows.

```
$ analyze/crash lockmgr.dmp

VAX/VMS System analyzer
Dump taken on 25-JUN-1988 19:18:15.86
LOCKMGRERR, Error detected by Lock Manager

SDA> read sys$system:sysdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SYSDEF.STB;1
SDA> read sys$system:scsdef.stb
%SDA-I-READSYM, reading symbol table  SYS$COMMON:[SYSEXE]SCSDEF.STB;1
SDA>!
SDA>!Was the structure defined in the predefined symbol tables?
SDA> format/type=lckmsg @R2
%SDA-E-NOSYMBOLS, no "LCKMSG" symbols found to format this block
SDA>!Not defined by symbol tables.
SDA> spawn

$!
$! Check the public libraries
$ search sys$share:starlet.req module,lckmsg/match=and
%SEARCH-I-NOMATCHES, no strings matched
$!
$ search sys$share:lib.req module,lckmsg/match=and
%SEARCH-I-NOMATCHES, no strings matched
$!
$! Not in public libraries
$! SDL file cannot be located or does not exist
$!
$! Must manually create our own structure
$ create lckmsg.mar
.macro $lckmsgdef                   ;Define the Macro
lckmsg$l_ppd_flink  ==-32           ;The numeric value is the byte offset
lckmsg$l_ppd_blink  ==-28           ; into the structure
lckmsg$b_ppd_swflag ==-21
lckmsg$b_ppd_type   ==-22
lckmsg$w_ppd_size   ==-24
lckmsg$b_ppd_flags  ==-17
lckmsg$b_ppd_opcode ==-18
lckmsg$b_ppd_status ==-19
lckmsg$b_ppd_port   ==-20
lckmsg$W_ppd_mtype  ==-14
lckmsg$W_ppd_msglen ==-16
lckmsg$W_scs_credit ==-10
lckmsg$W_scs_msgtyp ==-12
lckmsg$L_scs_source == -8
lckmsg$L_scs_dest   == -4
```

```
lckmsg$w_s_seq      ==  0          ;R2 will point to here
lckmsg$w_ack_seq    ==  2
lckmsg$l_rspid      ==  4
lckmsg$b_lck        ==  8
lckmsg$b_facility   ==  9
lckmsg$w_func       == 10
lckmsg$l_memseq     == 12
lckmsg$l_loc_lkid   == 16
lckmsg$l_rem_lkid   == 20
.endm
$lckmsgdef    ;Invoke the Macro
.end
^Z
$ macro lckmsg.mar
$ link/symbol_table/noexecutable lckmsg.obj
%LINK-W-USRTFR, image NL:[].EXE; has no user transfer address
$ logout
SDA>!Read in the newly created LCKMSG structure definition
SDA> read lckmsg.stb
%SDA-I-READSYM, reading symbol table  SYS$SYSROOT:[SYSMGR]LCKMSG.STB;1
SDA> format/type=lckmsg @R2
80412280   LCKMSG$L_PPD_FLINK      FFE643A0
80412284   LCKMSG$L_PPD_BLINK      FFE643A0
80412288   LCKMSG$W_PPD_SIZE           0090
8041228A   LCKMSG$B_PPD_TYPE         3C
8041228B   LCKMSG$B_PPD_SWFLAG     00
8041228C   LCKMSG$B_PPD_PORT             0C
8041228D   LCKMSG$B_PPD_STATUS         00
8041228E   LCKMSG$B_PPD_OPCODE       22
8041228F   LCKMSG$B_PPD_FLAGS      00
80412290   LCKMSG$W_PPD_MSGLEN         0079
80412292   LCKMSG$W_PPD_MTYPE      0004
80412294   LCKMSG$W_SCS_MSGTYP         000A
80412296   LCKMSG$W_SCS_CREDIT     0001
80412298   LCKMSG$L_SCS_SOURCE     7A11000A
8041229C   LCKMSG$L_SCS_DEST       7A110005
804122A0   LCKMSG$W_S_SEQ              1271
804122A2   LCKMSG$W_ACK_SEQ        0FC1
804122A4   LCKMSG$L_RSPID          6F3E0001
804122A8   LCKMSG$B_LCK                 02
804122A9   LCKMSG$B_FACILITY           07
804122AA   LCKMSG$W_FUNC           0000
804122AC   LCKMSG$L_MEMSEQ         00000002
804122B0   LCKMSG$L_LOC_LKID       000F012F
804122B4   LCKMSG$L_REM_LKID       0053025E
SDA>
```

A.5 Data Type Naming Conventions

The following information is extracted from the Internals and Data Structures Manual, VMS Version 5.2-Appendix D, Table D.1. Field format is *struc$X_fieldname*. Letter substitution for 'X' and the data types indicated are as described in Table A–1:

Table A–1: Data Type Definitions

Letter	Data Type/Usage
A	Address
B	Byte Integer
C	Character / Constant
D	Double Precision Floating
E	Reserved to Digital
F	Single Precision Floating
G	G_floating-point
H	H_floating-point
I	Reserved for Integer extension
J	Reserved to customer for escape to other codes
K	Constant (Preferred over C) (may indicate value or location of field within structure)
L	Longword Integer
M	Field Mask (pattern of bits)
N	Numeric String (byte form)(ascii numbers)
O	Reserved to Digital as an escape to other codes
P	Packed String (compressed data format)
Q	Quadword Integer
R	Reserved for Record structure
S	Field Size (size of field/structure)
T	Character String (Text) (ascii characters)
U	Smallest Unit of Addressable Storage
V	Field Position - VAX MACRO (bit location within specified field) Field reference - BLISS
W	Word Integer
X	Context-dependent
Y	Context-dependent
Z	Unspecified or Nonstandard
$Ax	Address of data whose size is specified by x (erl$al_bufaddr)
$Gx	Address of data whose size is specified by x (exe$gl_scb)

Appendix B

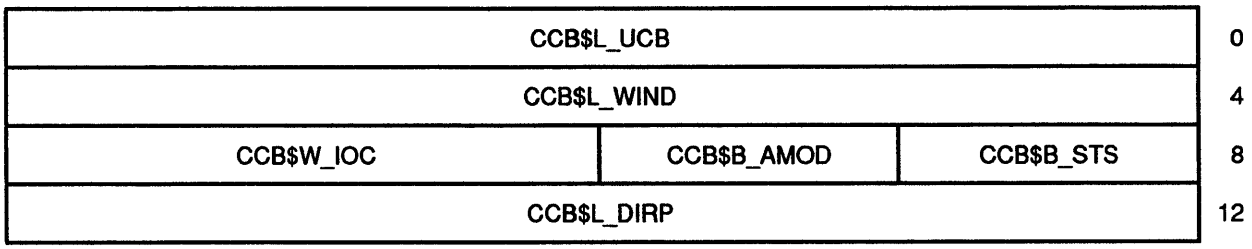
Data Structures

This appendix displays in detail the major VMS data structures described in this book.

Variations of a few of these can be found in other references; however, as of this writing, those other references did not describe fields and flags specific to DUDRIVER and the VMS based MSCP server. The descriptions provided here not only reflect the traditional use of these structures, but also the specifics relevant to the disk class driver and the MSCP server.

In addition to the explanations provided for the fields and flags, additional descriptive information accompanies many of the data structures. This information is intended to enhance some of the explanations given in this book, as well as provide further useful details.

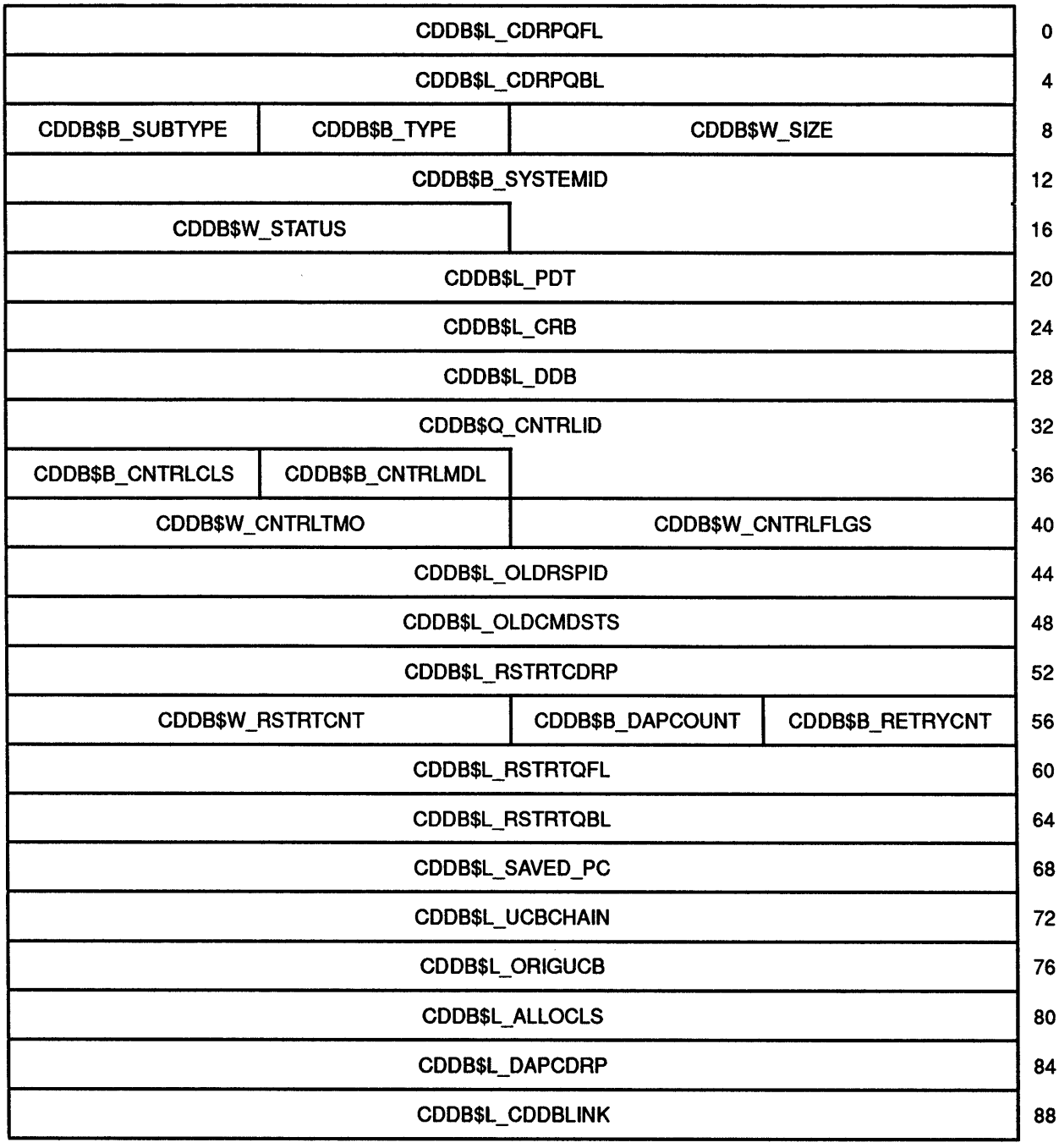
B.1 CCB - Channel Control Block



Field Name	Description and Flags
CCB$L_UCB	Address of UCB associated with device to which I/O channel has been assigned.
CCB$L_WIND	Address of window control block (WCB). A WCB is created when a file is accessed on the I/O channel with which this CCB is associated. The WCB provides virtual to logical block mapping information for the file.
CCB$B_STS	I/O channel status.
The following fields are defined within CCB$B_STS:	
CCB$V_AMB	Mailbox associated with channel (bit 0)
CCB$V_IMGTMP	Image temporary (bit 1)
CCB$V_RDCHKDON	Read protection check completed (bit 2)
CCB$V_WRTCHKDON	Write protection check completed (bit 3)
CCB$V_LOGCHKDON	Logical I/O access check done (bit 4)
CCB$V_PHYCHKDON	Physical I/O access check done (bit 5)
CCB$B_AMOD	Access mode plus 1 of the process at the time the I/O channel was assigned. When this field contains a 0, then this CCB is not in use.
CCB$W_IOC	Number of outstanding I/O requests on this I/O channel. This field is incremented by the $QIO system service code immediately after the IRP representing an I/O request has been allocated but not yet initialized. This field is decremented by the kernel AST queued to the process by I/O postprocessing upon completion of an I/O request.

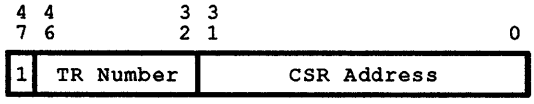
Field Name	Description and Flags
CCB$L_DIRP	Address of IRP for requested deaccess of the I/O channel with which this CCB is associated. A number of I/O requests can be pending concurrently on the same I/O channel. If the process which owns the channel issues I/O request to deaccess the device, the deaccess request is held until all other outstanding I/O requests are processed.

B.2 CDDB - Class Driver Data Block



CDDB$W_WTUCBCTR	CDDB$B_STS2	CDDB$B_FOVER_CTR	92
CDDB$W_CPYSEQNUM	CDDB$B_CHVRSN	CDDB$B_CSVRSN	96
CDDB$L_MAXBCNT			100
CDDB$L_CTRLTR_MASK			104
CDDB$W_RSVD4	CDDB$W_LOAD_AVAIL		108
CDDB$L_PERMCDRP			112

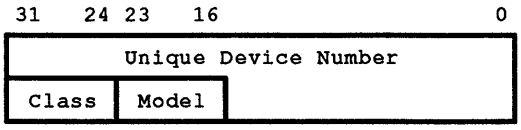
Field Name	Description and Flags
CDDB$L_CDRPQFL	Outstanding (i.e. active) CDRP queue forward link.
CDDB$L_CDRPQBL	Outstanding (i.e. active) CDRP queue backward link.
CDDB$W_SIZE	Size of this data structure.
CDDB$B_TYPE	Major structure type for class driver. DUDRIVER sets this field to contain the value of the symbol DYN$C_ CLASSDRV when the CDDB is created.
CDDB$B_SUBTYPE	Structure subtype field. DUDRIVER sets this field to contain the value of the symbol DYN$C_ CD_CDDB when the CDDB is created.
CDDB$B_SYSTEMID	48-bit system ID field. For a VAX host (which "looks like" a controller by virtue of running the VMS MSCP server), this is the SCSSYSTEMID assigned to it when it was installed. For an HSC, this is the value of the ID parameter which is also assigned at installation time. for a local controller handled by PUDRIVER, this quantity is constructed by PUDRIVER as follows:



CXN-000B-03

Field Name	Description and Flags
CDDB$W_STATUS	Status Flags.
The following fields are defined within CDDB$W_STATUS:	
CDDB$V_SNGLSTRM	Single stream processing of CDRPs (bit 0)
CDDB$V_IMPEND	Immediate command pending (bit 1)
CDDB$V_INITING	Initializing connection with MSCP server in controller (bit 2)
CDDB$V_RECONNECT	Reconnecting to MSCP server (bit 3)
CDDB$V_RESYNCH	Reconnect with MSCP server initiated by class driver (bit 4)
CDDB$V_POLLING	Currently polling for units (bit 5)
CDDB$V_ALCLS_SET	Allocation class has been set (bit 6)
CDDB$V_NOCON	Currently no connection with MSCP server in controller (bit 7)
CDDB$V_RSTRTWAIT	Waiting to restart next CDRP (bit 8)
CDDB$V_QUORLOST	CNXMAN quorum lost processing (bit 9)
CDDB$V_DAPBSY	DAP CDRP currently in use (bit 10 (A hex))
CDDB$V_2PBSY	Failover fork block busy (bit 11 (B hex))
CDDB$V_BSHADOW	Controller uses "bundled" Shadowing (bit 12 (C hex))
CDDB$V_DISABLED	Controller not in use by class driver action (bit 13 (D hex))
CDDB$V_PATHMOVE	Closing connection for port load balance (bit 14 (E hex))
CDDB$V_PRMBSY	Permanent CDRP in use (bit 15 (F hex))
CDDB$L_PDT	Port Descriptor Table address.
CDDB$L_CRB	CRB address.
CDDB$L_DDB	Address of first DDB in list of DDBs linked to this CDDB by means of the CONLINK field in each subsequent DDB.
CDDB$Q_CNTRLID	Controller ID. This 64-bit quantity is returned in the end message from the SET CONTROLLER CHARACTERISTICS command issued by DUDRIVER during controller initialization. It consists of three fields: Class (CDDB$B_CNTRLCLS) Identifies the type of subsystem. For DSA controllers and VAXes running the VMS MSCP server software (and thus look like a controller), this field contains a 1 (i.e. mass storage controller). Model (CDDB$B_CNTRLMDL) Identifies the exact model of the subsystem within this class. Some of the common model numbers as of this writing are as follows:

Field Name	Description and Flags
The following decimal values are defined for the Controller Model:	
000 (00 hex)	Unknown controller model
001 (01 hex)	HSC50
002 (02 hex)	UDA50
004 (04 hex)	VMS Emulator (software MSCP server)
013 (0D hex)	KDA50
018 (12 hex)	KDB50
021 (15 hex)	DSSI disk
027 (1B hex)	KDM70
032 (20 hex)	HSC70
033 (21 hex)	HSC40
034 (22 hex)	HSC60
035 (23 hex)	HSC90
097 (61 hex)	RF71
101 (65 hex)	RF72
102 (66 hex)	RF73



CXN-000B-04

CDDB$W_CNTRLFLGS — Controller flags returned in end message corresponding to an MSCP SET CONTROLLER CHARACTERISTICS command.

Field Name	Description and Flags
The following fields are defined within CDDB$W_CNTRLFLGS field:	
MSCP$V_CF_576	Controller supports disks formatted with 576-byte sectors. (bit 0)
MSCP$V_CF_SHADW	Controller supports volume shadowing (bit 1)
MSCP$V_CF_MLTHS	Multiple host controller (bit 2)
MSCP$V_CF_THIS	Error log messages related to commands issued by this host should be sent to this host (bit 4)
MSCP$V_CF_OTHER	Error log messages related to commands issued by other hosts should be sent to this host (bit 5)
MSCP$V_CF_MISC	Error log messages which do not relate to a specific command should be sent to this host (bit 6)
MSCP$V_CF_ATTN	Attention messages should be sent to this host (bit 7)
MSCP$V_CF_LOAD	Controller returns load available information (bit 13 (D hex))
MSCP$V_CF_EDCRP	Data encrypt/decrypt supported (bit 14 (E hex))
MSCP$V_CF_REPLC	Controller handles bad block replacement for disks connected to controller (bit 15 (F hex))
MSCP$V_CF_SRT	Segemented Record Transfer (bit 15 (F hex))
CDDB$W_CNTRLTMO	Controller timeout in seconds. This field is set up by DUDRIVER when it establishes a connection with the MSCP server in the controller. The quantity stored here determines how often the CRB associated with the controller expires and invokes the class driver timeout mechanism routine. In DUDRIVER, this routine is *DU$TMR*.
CDDB$L_OLDRSPID	RSPID of oldest outstanding MSCP command. This field is valid only when the RSPID it contains matches the RSPID of the CDRP at the head of this CDDB's outstanding (i.e. active) CDRP queue.
CDDB$L_OLDCMDSTS	Latest MSCP command status for command corresponding to RSPID stored in OLDRSPID field. This field is used to determine if the controller is making progress with the oldest outstanding command.
CDDB$L_RSTRTCDRP	Address of only active CDRP if in single stream mode.
CDDB$B_RETRYCNT	Number of retries remaining for currently active CDRP if in single stream mode.
CDDB$B_DAPCOUNT	Number of remaining calls to DU$TMR before DU$TMR actually initiates DAP processing.
CDDB$W_RSTRTCNT	Number of resynch or connection errors since VMS booted on this host.

Field Name	Description and Flags
CDDB$L_RSTRTQFL	Forward link of queue wherein CDRPs are accumulated, sorted, and selected for resubmission to the controller after reestablishing the connection with the MSCP server in that controller.
CDDB$L_RSTRTQBL	Backward link of CDRP restart queue described above.
CDDB$L_SAVED_PC	Saved PC on internal subroutine calls.
CDDB$L_UCBCHAIN	Head of list of UCBs chained to this CDDB.
CDDB$L_ORIGUCB	Address of "original UCB" (i.e. the boot UCB) if unchained.
CDDB$L_ALLOCLS	Device allocation class.
CDDB$L_DAPCDRP	Address of CDRP used for Determine Access Paths processing.
CDDB$L_CDDBLINK	Address of next CDDB in list of CDDBs available to class driver.
CDDB$B_FOVER_CTR	Counter of reconnect intervals per failover try.
CDDB$B_STS2	Further Status Bits.
CDDB$W_WTUCBCTR	Counter of UCBs waiting for mount verification to finish so that single stream processing of CDRPs may begin.
CDDB$B_CSVRSN	Controller microcode version.
CDDB$B_CHVRSN	Controller hardware version.
CDDB$W_CPYSEQNUM	Base value of IO$_COPYSHAD sequence number.
CDDB$L_MAXBCNT	Maximum byte count for the controller associated with this CDDB.
CDDB$L_CTRLTR_MASK	Mask of controller letters (ddCu:) used by this controller.
CDDB$L_LOAD_AVAIL	Load available from MSCP server.
CDDB$L_PERMCDRP	This is the beginning of the permanent IRP/CDRP pair allocated contiguous with the CDDB. Since the CDRP is actually an extension of the IRP, this marks the beginning of the IRP component of the pair. Following this first extension of the basic CDDB is a second disk class driver extension: the DAP IRP/CDRP pair.

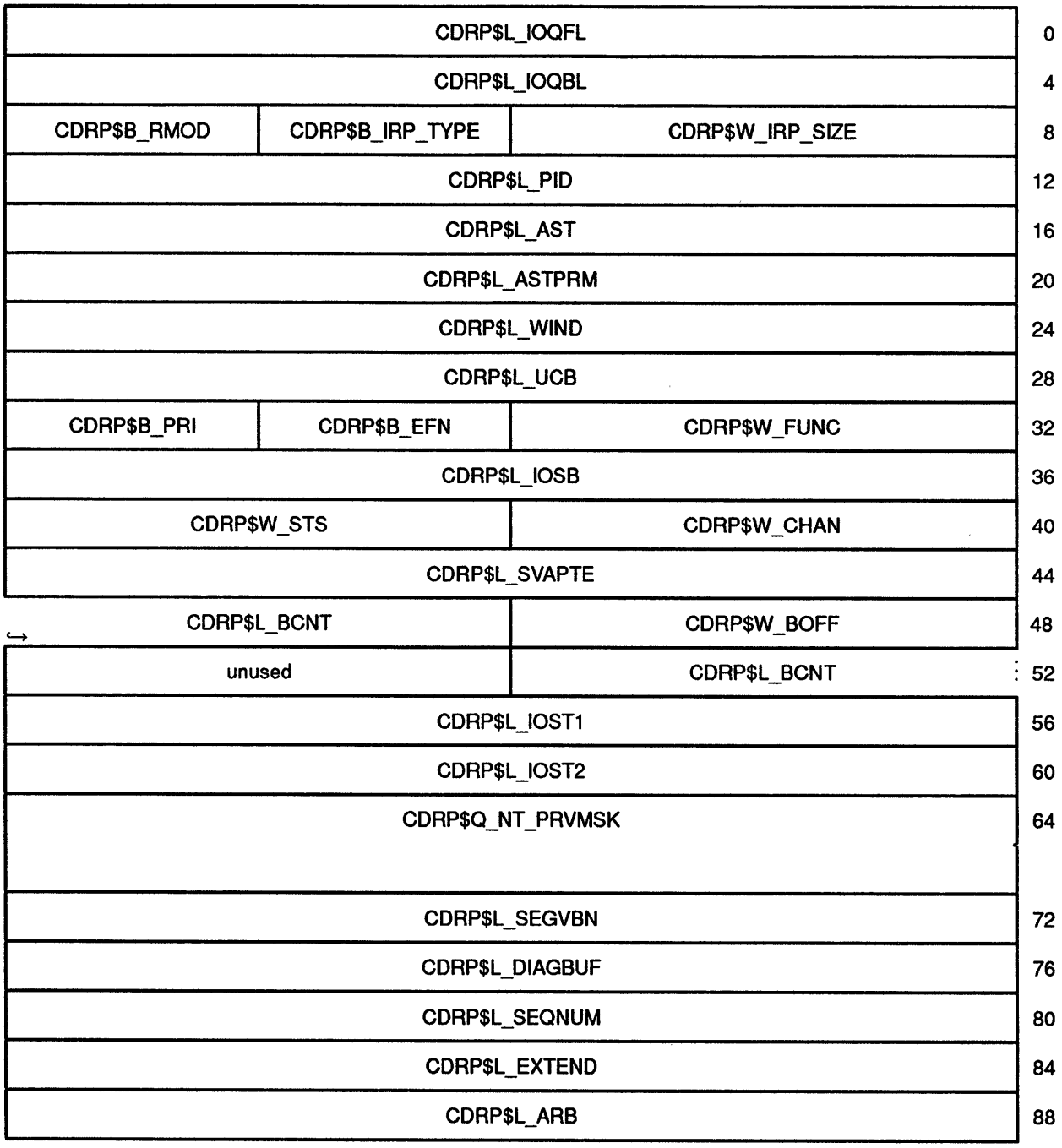
NOTE

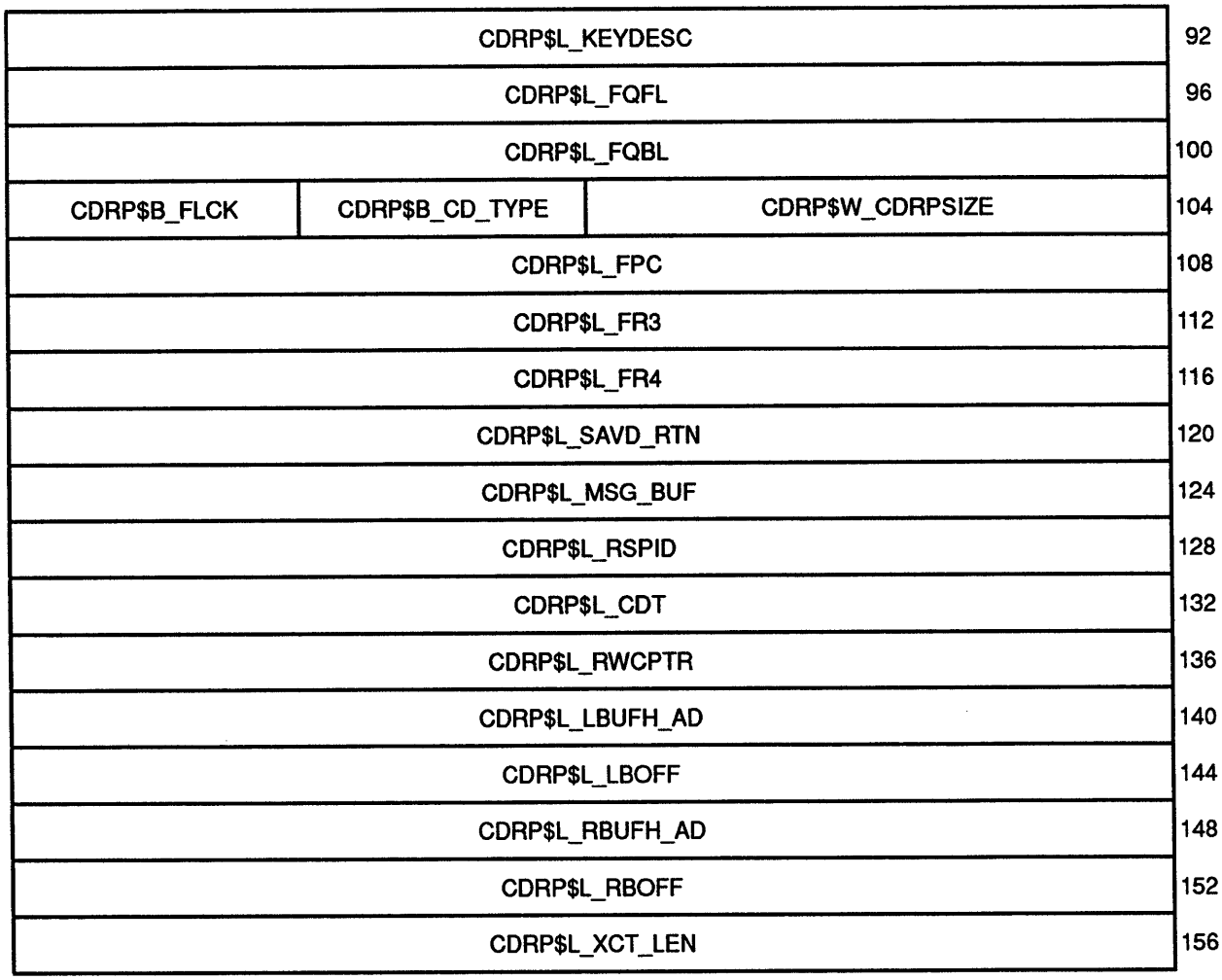
The fields that follow are actually offsets relative to the beginning of the CDDB and into the permanent IRP/CDRP which is contiguous with the end of the basic CDDB.

As of this writing, the values of these symbolic offsets are not available by default to SDA. However, they are defined by the macro $DUTUDEF which can be found in module DUTUMAC of VMS.

Field Name	Description and Flags
CDDB$A_PRMIRP	Beginning of the IRP component of the permanent IRP/CDRP pair allocated as a disk class driver extension at the end of the basic CDDB. This symbol has the same value as CDDB$L_PERMCDRP.
CDDB$L_PRMUCB	This is an alternate name for the IRP$L_UCB field in the IRP component of the permanent IRP/CDRP pair.
CDDB$L_CANCLQFL	Forward link of IRPs representing requests to cancel one or more I/O requests. This field overlays the IRP$L_ABCNT field in the IRP component of the permanent IRP/CDRP pair.
CDDB$L_CANCLQBL	Backward link of IRPs representing requests to cancel one or more I/O requests. This field overlays the IRP$L_OBCNT field in the IRP component of the permanent IRP/CDRP pair.
CDDB$A_PRMCDRP	Permanent CDRP.
CDDB$L_CDT	This field overlays the IRP$L_CDT field in the permanent IRP/CDRP pair.
CDDB$A_DAPIRP	This symbol marks the beginning of the IRP component of the DAP IRP/CDRP pair, the second disk class driver CDDB extension.
CDDB$L_DAPUCB	This field overlays the IRP$L_IRP field in the IRP component of the DAP IRP/CDRP pair.
CDDB$A_DAPCDRP	CDRP component of DAP IRP/CDRP begins here.
CDDB$L_DAPCDT	This field overlays the IRP$L_CDT field of the DAP IRP/CDRP pair.
CDDB$A_2PFKB	Beginning of fork block within DAP IRP/CDRP used for unit failover by the disk class driver. This fork block overlaps the last few longwords of the IRP component of the DAP IRP/CDRP, beginning with the IRP$L_ABCNT byte. This is permissable since the longwords aren't used anyway for their normal purpose; so this was done to save nonpaged pool and to use previously unused space in this CDDB. There are two longwords which have their own symbolic names relative to the beginning of this fork block: FKB2P$L_SAVD_RTN - Saved return PC for routine DUTU$FIND_DDB FKB2P$L_SAVD_UCB - Saved UCB address for routine DUTU$MOVE_IODB

B.3 CDRP - Class Driver Request Packet



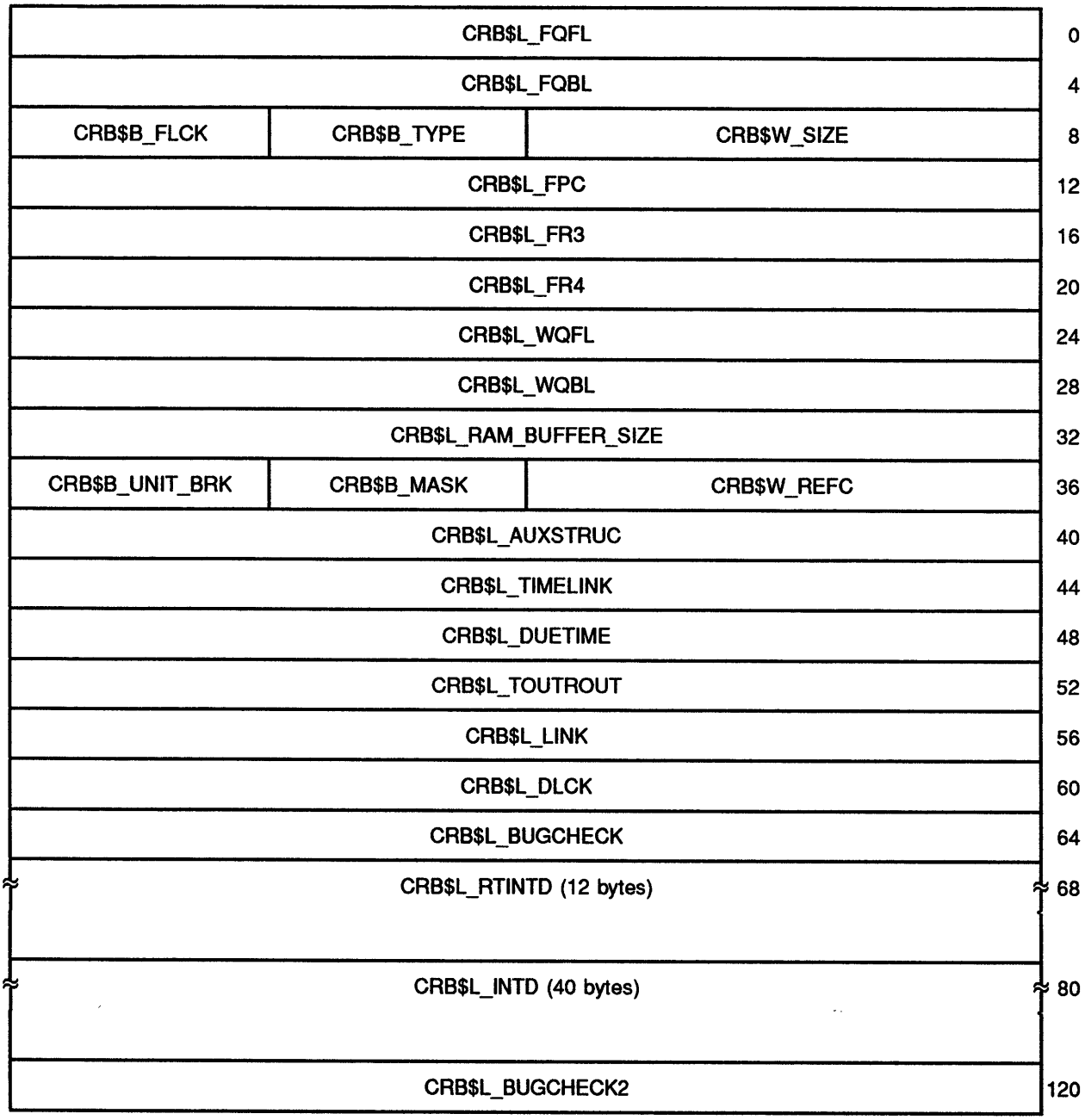


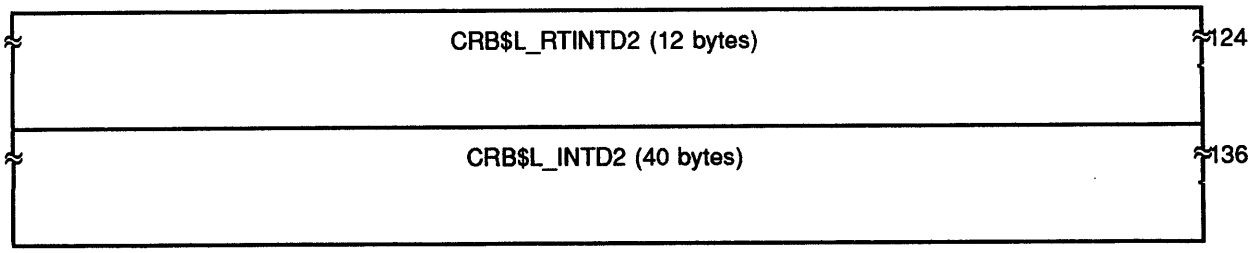
Field Name	Description and Flags
CDRP$L_IOQFL	I/O Queue Forward Link
CDRP$L_IOQBL	I/O Queue Backward Link
CDRP$W_IRP_SIZE	Size of IRP in Bytes
CDRP$B_IRP_TYPE	Structure Type for IRP
CDRP$B_RMOD	Access Mode of Request
CDRP$L_PID	Process ID of Requesting Process
CDRP$L_AST	Address of AST Routine
CDRP$L_ASTPRM	AST Parameter

Field Name	Description and Flags
CDRP$L_WIND	Address of Window Block
CDRP$L_UCB	Address of Device UCB
CDRP$W_FUNC	I/O Function Code and Modifiers
CDRP$B_EFN	Event Flag Number and Event Group
CDRP$B_PRI	Base Priority of Requesting Process
CDRP$L_IOSB	Address of I/O Status Double Longword
CDRP$W_CHAN	Process I/O Channel Number
CDRP$W_STS	Request Status
CDRP$L_SVAPTE	System Virtual Address of First PTE
CDRP$W_BOFF	Byte Offset in First Page
CDRP$L_BCNT	Byte Count of Transfer
CDRP$L_IOST1	First I/O Status Longword (for I/O post)
CDRP$L_IOST2	Second I/O Status Longword
CDRP$Q_NT_PRVMSK	Privilege Mask for DECNET
CDRP$L_SEGVBN	Virtual Block Number of Current Segment
CDRP$L_DIAGBUF	Diagnostic Buffer Address
CDRP$L_SEQNUM	Sequence Number
CDRP$L_EXTEND	Address of IRPE
CDRP$L_ARB	Access Rights Block Address
CDRP$L_KEYDESC	Address of Encryption Key Descriptor
CDRP$L_FQFL	Fork queue forward link. The class driver request may be suspended waiting for a resource or an event to occur. In such situations, the CDRP representing the request is queued to a data structure via this queue linkage field. An example would be a DUDRIVER request waiting for the end message corresponding to an MSCP command sent to the disk server in an HSC. The CDRP representing that request would be queued to the CDDB for the HSC.
CDRP$L_FQBL	Fork queue backward link.
CDRP$W_CDRPSIZE	Size of this data structure. The size of a CDRP can vary, depending on the presence of CDRP extensions. CDRPs can be allocated as part of a larger data structure, such as an IRP. The permanent CDRP defined at the end of a CDDB is an example of a CDRP defined in a data structure which is not an IRP. In this case the CDRPSIZE field is set to a negative number, the offset from the base of the CDRP to the base of the CDDB.

Field Name	Description and Flags
CDRP$B_CD_TYPE	Type of data structure. This field is loaded with the value of the symbol DYN$C_CDRP when the CDRP is created.
CDRP$B_FLK	Fork Lock.
CDRP$L_FPC	Fork PC. When a class driver request represented by a CDRP is suspended waiting for a resource to come available or an event to occur, this field contains the PC at which to resume the request when its wait is satisfied.
CDRP$L_FR3	Fork R3. When a class driver request represented by a CDRP is suspended waiting for a resource to come available or an event to occur, this field contains the quantity to be restored to register R3 when the request is resumed.
CDRP$L_FR4	Fork R4. When a class driver request represented by a CDRP is suspended waiting for a resource to come available or an event to occur, this field contains the quantity to be restored to register R4 when the request is resumed.
CDRP$L_SAVD_RTN	Save return address. Various routines use this field to save the return address of the routine's caller. It is also sometimes used as a scratch area.
CDRP$L_MSG_BUF	Message buffer address. This field is used to hold the address of an SCS message buffer which is allocated but not yet queued to the port for transmission. When given a $QIO request, DUDRIVER allocates an SCS message buffer in which to build the MSCP command it will send to an "MSCP speaking" controller. The address of that buffer is kept here while DUDRIVER is building the MSCP command. When the CDRP is passed to the SCS/PPD layers, this field is set to 0 once the message has been sent.
CDRP$L_RSPID	Response ID. When nonzero, the local SYSAP sending a message expects a response from the SYSAP to which it is sending the message. This quantity is used by the SCS layer to associated the response, when received, with the CDRP. Then, using the FPC, FR3, and FR4 fields, the context of the local SYSAP can be resumed when the response arrives.
CDRP$L_CDT	Address of the CDT describing the connection between the local SYSAP which issued the request represented by this CDRP, and the remote SYSAP with which it is communicating via SCS.
CDRP$L_RWCPTR	Address of the resource wait count field, UCB$L_RWAITCNT, in a disk or tape class driver UCB. Disk and tape class driver I/O requests for a unit are stalled if the RWAITCNT field in the unit's UCB is nonzero.
	Block Transfer Extension
CDRP$L_LBUFH_AD	Local BUFfer Handle ADdress
CDRP$L_LBOFF	Local Byte OFFset
CDRP$L_RBUFH_AD	Remote BUFfer Handle ADdress
CDRP$L_RBOFF	Remote Byte OFFset
CDRP$L_XCT_LEN	Transfer length in bytes

B.4 CRB - Channel Request Block



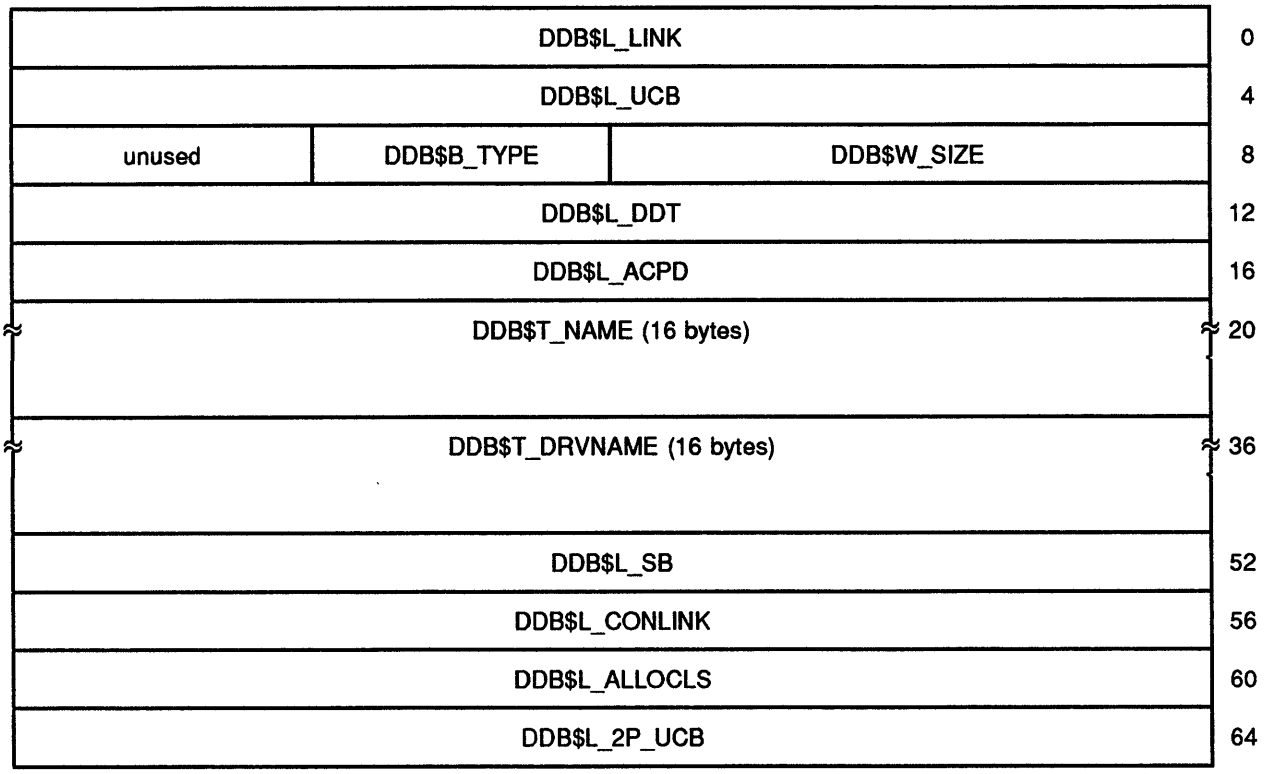


Field Name	Description and Flags
CRB$L_FQFL	Fork Queue Forward Link
CRB$L_FQBL	Fork Queue Backward Link
CRB$W_SIZE	Size of CRB in Bytes
CRB$B_TYPE	Structure Type for CRB. The value of the symbol DYN$C_CRB is stored in this field when the CRB is created.
CRB$B_FLCK	Fork Lock Number
CRB$L_FPC	Fork PC
CRB$L_FR3	Fork R3
CRB$L_FR4	Fork R4
CRB$L_WQFL	Controller data channel wait queue forward link. A channel wait queue contains addresses of driver fork blocks that contain the context of suspended driver forks waiting to gain control of a controller data channel.
CRB$L_WQBL	Controller data channel wait queue backward link.
CRB$L_RAM_ BUFFER_SIZE	Size of RAM buffer
CRB$W_REFC	UCB reference count. The number of UCBs corresponding to devices attached to the controller with which this CRB is associated.
CRB$B_MASK	Channel allocation mask. Also known as the controller status mask. As of this writing, there is only one flag defined: CRB$V_BSY - Channel is busy (not free)
CRB$B_UNIT_BRK	Break bits for lines.

Field Name	Description and Flags
CRB$L_AUXSTRUC	Address of auxiliary data structure used by the device driver to store special controller information. In the case of the disk class driver, DUDRIVER, this field contains the address of the CDDB associated with this controller.
CRB$L_TIMELINK	Forward link in list of CRBs waiting for periodic wakeups. This field contains the address of the TIMELINK field of the next CRB in the list. The last CRB in the list has this field set to 0. Location IOC$GL_CRBTMOUT is the listhead. Routine EXE$TIMEOUT in module TIMESCHDL tends to VMS's periodic "once a second" tasks. One of these tasks is to scan this list for CRBs which have timed out (see DUETIME field). If a timed out CRB is found, the routine whose address is in the TOUTROUT field is called.
CRB$L_DUETIME	Time in seconds when this CRB expires and the periodic wakeup associated with this CRB is to be delivered. Routine EXE$TIMEOUT compares the content of this field with the content of EXE$GL_ABSTIM. If DUETIME is greater than ABSTIM, then this CRB has not yet come due. If, however, DUETIME is less than or equal to ABSTIM, then this CRB has timed out and the wakeup is delivered by calling the routine whose address is in the TOUTROUT field.
CRB$L_TOUTROUT	This field contains the address of the routine to be called when this CRB times out. One of the tasks this routine must do is to reset the DUETIME field to the next wakeup time. In the case of DUDRIVER, this routine is DU$TMR. And DU$TMR "knows" with which controller the wakeup is associated since it has the address of the CDDB associated with that controller in the AUXSTRUC field of this CRB.
CRB$L_LINK	Address of secondary CRB (for MASSBUS devices only).
CRB$L_DLCK	Address of Device Spinlock
CRB$L_BUGCHECK	Address of ILLQBUSCFG Bug_Check
CRB$L_RTINTD	2ND Q-22 BUS Multi-level Intr Dispatch Code Start

Field Name	Description and Flags
CRB$L_INTD	Interrupt transfer vector. The Driver Prologue Table in each driver for an interrupting device specifies the address of an interrupt service routine as well as other information to be stored in this 36-byte region. However, no devices directly interrupt the disk class driver. DUDRIVER exchanges messages with the controllers for its disks. These messages are passed down through the software to a port driver for transmission to the controller. When a message is received from a controller. the interrupt is received by a port driver (e.g. PADRIVER, PUDRIVER, ...). The port driver then passes the message back up through the software to DUDRIVER. Consequently, DUDRIVER makes makes almost no use of this region. See source listings of PADRIVER and PUDRIVER for examples of what is stored here and its use. See also the manual entitled "Writing a Device Driver for VAX/VMS" to obtain details of what is in this region for drivers that do make use of it.
CRB$L_BUGCHECK2	Address of 2nd ILLQBUSCFG Bug_Check
CRB$L_RTINTD2	2ND Q-22 BUS Multi-level Intr Dispatch Code Start
CRB$L_INTD2	Second 36-byte interrupt transfer vector for devices with multiple interrupt vectors.

B.5 DDB - Device Data Block

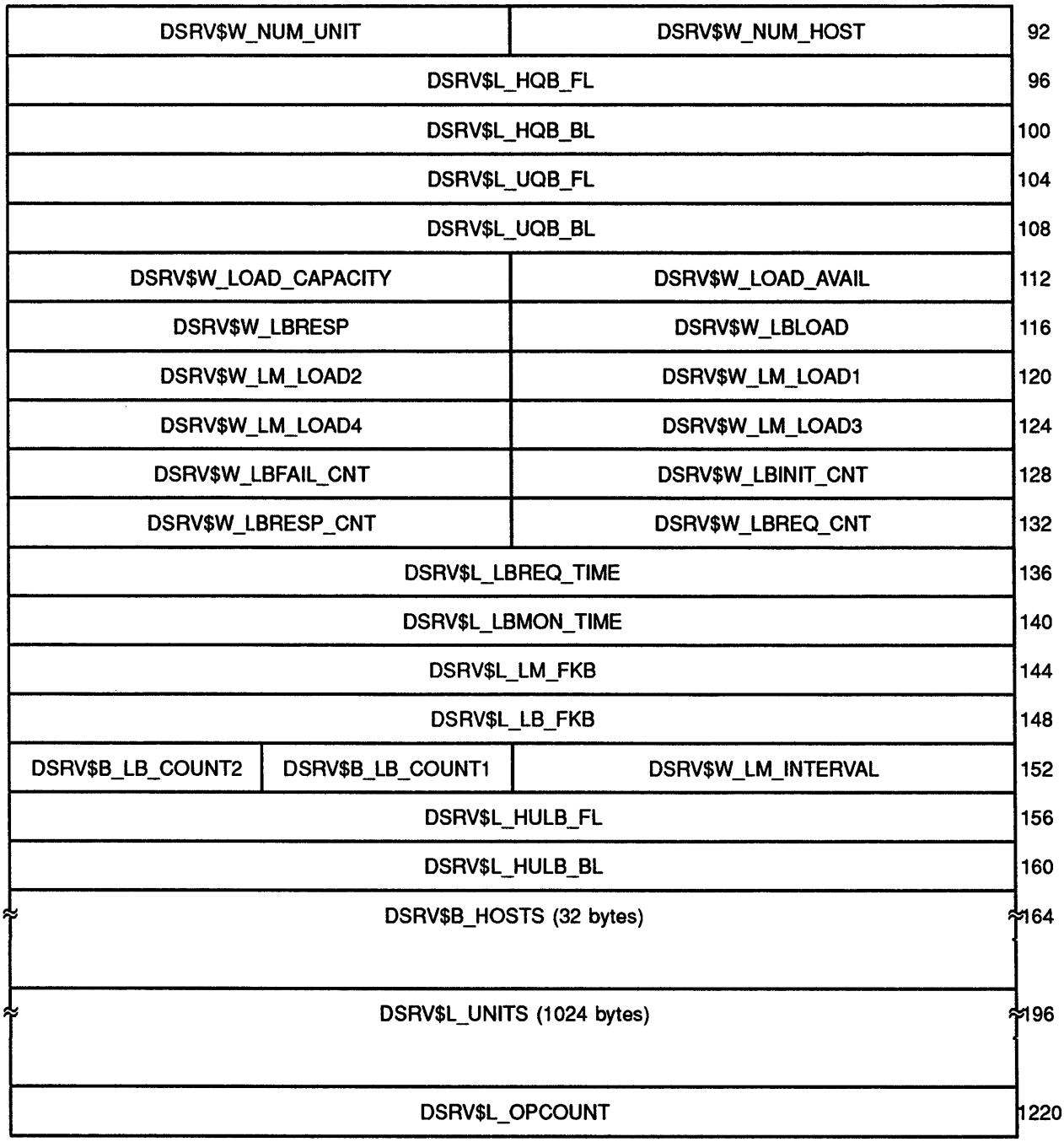


Field Name	Description and Flags
DDB$L_LINK	Address of next DDB in chain of DDBs linked to an SB. A zero in this field indicates that this is the last DDB in the chain.
DDB$L_UCB	Address of first UCB in chain of UCBs linked to this DDB.
DDB$W_SIZE	Size of this data structure.
DDB$B_TYPE	Type of data structure. This field is loaded with the value of the symbol DYN$C_DDB when the DDB is created.
DDB$L_DDT	Address of DDT. The Driver Prologue Table of each device driver specifies the address to be loaded into this field when the DDB is created.

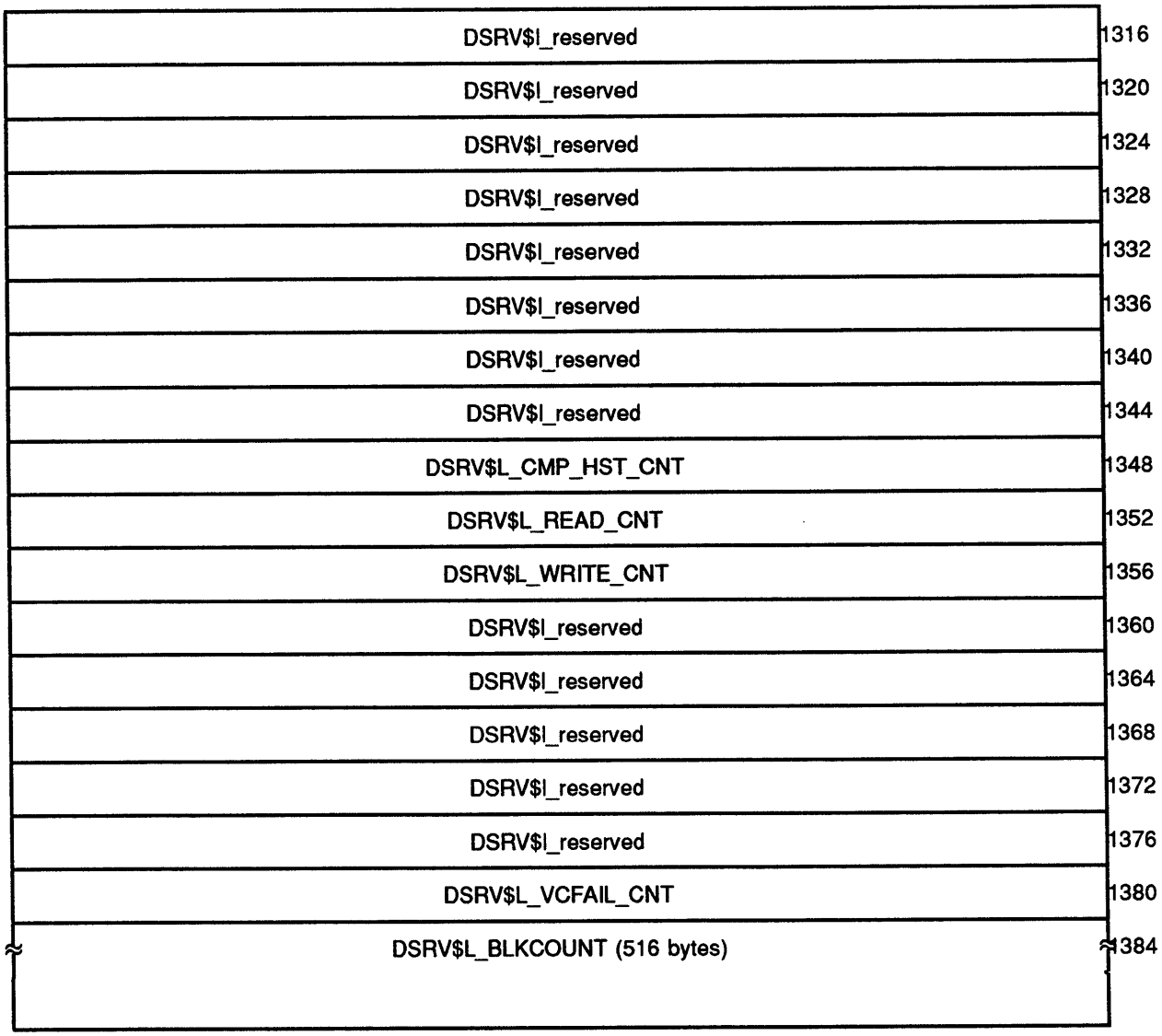
Field Name	Description and Flags
DDB$L_ACPD	Name of default ACP for controller. ACPs that control access to file-structured devices use the high order byte of this field, called DDB$B_ACPCLASS, to indicate the class of the file-structured device. Both the default ACP name and class of file-structured device (if there is one) are specified by the Driver Prologue Table. The default ACP specified by DUDRIVER's Driver Prologue Table is "F11". Values for the DDB$B_ACPCLASS field (the high order byte of the DDB$L_ACPD field) are: DDB$K_PACK - Standard disk pack (value 1) DDB$K_CART - Cartridge disk pack (value 2) DDB$K_SLOW - Floppy disk (value 3) DDB$K_TAPE - File-structured magnetic tape (value 4)
DDB$T_NAME	Generic name of devices attached to controller (e.g. DU, DJ, ...). The first byte of this 16-byte field is the number of characters in the generic name. The remainder of the field consists of a string of up to 15 characters.
DDB$T_DRVNAME	Name of device driver (e.g. DUDRIVER, DRDRIVER, ...). The first byte of this 16-byte field is the number of characters in the device driver name. The remainder of the field consists of a string of up to 15 characters.
DDB$L_SB	Address of System Block.
DDB$L_CONLINK	Address of next DDB in the connection subchain. For disks handled by DUDRIVER, this connection subchain consists of a list of DDBs linked into the CDDB. The listhead is the CDDB$L_DDB field in the CDDB.
DDB$L_ALLOCLS	Allocation class. If no allocation class is assigned, then this field contains a zero.
DDB$L_2P_UCB	Address of the first in a chain of UCBs on secondary path.

B.6 DSRV - Disk Server Structure

DSRV$L_FLINK			0
DSRV$L_BLINK			4
DSRV$B_SUBTYPE	DSRV$B_TYPE	DSRV$W_SIZE	8
DSRV$W_BUFWAIT		DSRV$W_STATE	12
DSRV$L_LOG_BUF_START			16
DSRV$L_LOG_BUF_END			20
DSRV$L_NEXT_READ			24
DSRV$L_NEXT_WRITE			28
DSRV$W_INC_HILIM		DSRV$W_INC_LOLIM	32
DSRV$W_EXC_HILIM		DSRV$W_EXC_LOLIM	36
DSRV$L_SRVBUF			40
DSRV$L_FREE_LIST			44
DSRV$L_AVAIL			48
DSRV$L_BUFFER_MIN			52
DSRV$L_SPLITXFER			56
DSRV$W_CFLAGS		DSRV$W_VERSION	60
DSRV$w_reserved		DSRV$W_CTIMO	64
DSRV$Q_CTRL_ID			68
DSRV$L_MEMW_TOT			76
DSRV$W_MEMW_MAX		DSRV$W_MEMW_CNT	80
DSRV$L_MEMW_FL			84
DSRV$L_MEMW_BL			88


DSRV$W_NUM_UNIT
DSRV$W_NUM_HOST
92
DSRV$L_HQB_FL
96
DSRV$L_HQB_BL
100
DSRV$L_UQB_FL
104
DSRV$L_UQB_BL
108
DSRV$W_LOAD_CAPACITY
DSRV$W_LOAD_AVAIL
112
DSRV$W_LBRESP
DSRV$W_LBLOAD
116
DSRV$W_LM_LOAD2
DSRV$W_LM_LOAD1
120
DSRV$W_LM_LOAD4
DSRV$W_LM_LOAD3
124
DSRV$W_LBFAIL_CNT
DSRV$W_LBINIT_CNT
128
DSRV$W_LBRESP_CNT
DSRV$W_LBREQ_CNT
132
DSRV$L_LBREQ_TIME
136
DSRV$L_LBMON_TIME
140
DSRV$L_LM_FKB
144
DSRV$L_LB_FKB
148
DSRV$B_LB_COUNT2
DSRV$B_LB_COUNT1
DSRV$W_LM_INTERVAL
152
DSRV$L_HULB_FL
156
DSRV$L_HULB_BL
160
DSRV$B_HOSTS (32 bytes)
164
DSRV$L_UNITS (1024 bytes)
196
DSRV$L_OPCOUNT
1220

DSRV$L_ABORT_CNT	1224
DSRV$L_GET_CMD_CNT	1228
DSRV$L_GET_UNT_CNT	1232
DSRV$L_SET_CON_CNT	1236
DSRV$l_reserved	1240
DSRV$l_reserved	1244
DSRV$l_reserved	1248
DSRV$L_AVAIL_CNT	1252
DSRV$L_ONLIN_CNT	1256
DSRV$L_SET_UNT_CNT	1260
DSRV$L_DET_ACC_CNT	1264
DSRV$l_reserved	1268
DSRV$l_reserved	1272
DSRV$l_reserved	1276
DSRV$l_reserved	1280
DSRV$L_ACCES_CNT	1284
DSRV$L_CMP_CON_CNT	1288
DSRV$L_ERASE_CNT	1292
DSRV$L_FLUSH_CNT	1296
DSRV$L_REPLC_CNT	1300
DSRV$l_reserved	1304
DSRV$l_reserved	1308
DSRV$l_reserved	1312



Field Name	Description and Flags
DSRV$L_FLINK	Not used since this structure is not in any queue.
DSRV$L_BLINK	Not used since this structure is not in any queue.
DSRV$W_SIZE	Size of this data structure.

Field Name	Description and Flags
DSRV$B_TYPE	Type of data structure. This field is initialized to contain the value of the symbol DYN$C_ DSRV when the MSCP server initializes.
DSRV$B_SUBTYPE	Data structure subtype. This field is initialized to contain the value of the symbol DYN$C_ DSRV_DSRV when the MSCP server is initialized.
DSRV$W_STATE	Current state of the server.

The following fields are defined within DSRV$W_STATE:	
DSRV$V_LOG_ENABLD	Logging is enabled. (bit 0)
DSRV$V_LOG_PRESENT	Logging code is present. (bit 1)
DSRV$V_PKT_LOGGED	A packet has been logged. (bit 2)
DSRV$V_PKT_LOST	One or more packets since last read. (bit 3)
DSRV$V_LBSTEP1	Load Balance Step One (bit 4)
DSRV$V_LBSTEP2	Load Balance Step Two (bit 5)
DSRV$V_LBEVENT	Load Balance Event (bit 6)
DSRV$V_HULB_DEL	HULB Deletion (bit 7)
DSRV$V_MON_ACTIVE	Load Monitor Fork Block Active (bit 8)
DSRV$V_LB_REQ	Load Balance Required (bit 9)
DSRV$V_CONFIG_WAIT	Waiting for STACONFIG to configure devices (bit 10 (A hex))

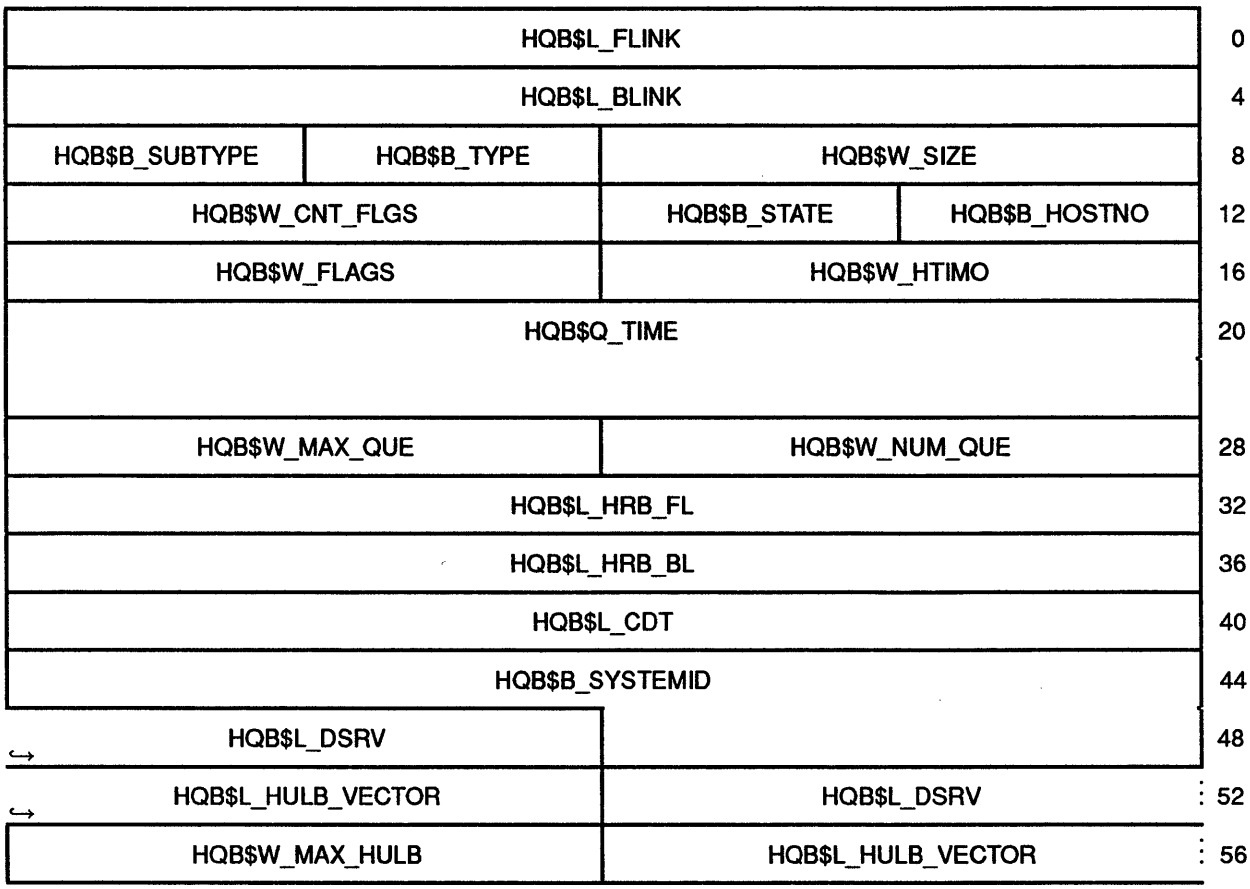
DSRV$W_BUFWAIT	I/Os that had to Wait
DSRV$L_LOG_BUF_START	Address of start of buffer.
DSRV$L_LOG_BUF_END	Address of end of buffer.
DSRV$L_NEXT_READ	Address of next packet to read.
DSRV$L_NEXT_WRITE	Address of next packet to write.
DSRV$W_INC_LOLIM	Low unit number to log.
DSRV$W_INC_HILIM	High unit number to log.
DSRV$W_EXC_LOLIM	Low unit number not to log.
DSRV$W_EXC_HILIM	High unit number not to log.
DSRV$L_SRVBUF	Address of preallocated pool of transfer buffers.
DSRV$L_FREE_LIST	Head of linked list of free transfer buffers in preallocated pool.
DSRV$L_AVAIL	Sum of Bytes Available in Buffer
DSRV$L_BUF_MIN	Minimum Transfer size based on buffer .

Field Name	Description and Flags
DSRV$L_SPLITXFER	Fragmented I/O Count.
DSRV$W_VERSION	Server Software Version
DSRV$W_CFLAGS	Controller Flags
The following fields are defined within DSRV$W_FLAGS:	
MSCP$V_CF_576	Controller supports disks formatted with 576-byte sectors. (bit 0)
MSCP$V_CF_SHADW	Controller supports volume shadowing (bit 1)
MSCP$V_CF_MLTHS	Multiple host controller (bit 2)
MSCP$V_CF_THIS	Error log messages related to commands issued by this host should be sent to this host (bit 4)
MSCP$V_CF_OTHER	Error log messages related to commands issued by other hosts should be sent to this host (bit 5)
MSCP$V_CF_MISC	Error log messages which do not relate to a specific command should be sent to this host (bit 6)
MSCP$V_CF_ATTN	Attention messages should be sent to this host (bit 7)
MSCP$V_CF_LOAD	Controller returns load available information (bit 13 (D hex))
MSCP$V_CF_EDCRP	Data encrypt/decrypt supported (bit 14 (E hex))
MSCP$V_CF_REPLC	Controller handles bad block replacement for disks connected to controller (bit 15 (F hex))
MSCP$V_CF_SRT	Segemented Record Transfer (bit 15 (F hex))
DSRV$W_CTIMO	Controller timeout.
DSRV$W_RESERVED	Reserved
DSRV$Q_CTRL_ID	Unique MSCP device identifier (controller ID).
DSRV$L_MEMW_TOT	Number of I/Os that had to wait
DSRV$W_MEMW_CNT	Number of requests in memory wait queue.
DSRV$W_MEMW_MAX	Most requests ever in memory wait queue.
DSRV$L_MEMW_FL	Forward link for head of I/O request memory wait queue.
DSRV$L_MEMW_BL	Backward link for head of I/O request memory wait queue.
DSRV$W_NUM_HOST	Count of hosts being served.
DSRV$W_NUM_UNIT	Count of disks being served.
DSRV$L_HQB_FL	Forward link for head of queue of Host Queue Blocks.
DSRV$L_HQB_BL	Backward link for head of queue of Host Queue Blocks.
DSRV$L_UQB_FL	Forward link for head of queue of Unit Queue Blocks.
DSRV$L_UQB_BL	Backward link for head of queue of Unit Queue Blocks.
DSRV$W_LOAD_AVAIL	Current load available

Field Name	Description and Flags
DSRV$W_LOAD_CAPACITY	Server load capacity
DSRV$W_LBLOAD	Target load for LB request
DSRV$W_LBRESP	Load available from other server
DSRV$W_LM_LOAD1	previous interval load 1
DSRV$W_LM_LOAD2	previous interval load 2
DSRV$W_LM_LOAD3	previous interval load 3
DSRV$W_LM_LOAD4	previous interval load 4
DSRV$W_LBINIT_CNT	Count of LB requests we have sent
DSRV$W_LBFAIL_CNT	Count of LB requests that failed
DSRV$W_LBREQ_CNT	Count of LB requests from other servers
DSRV$W_LBRESP_CNT	Count of LB requests we to which we responded
DSRV$L_LBREQ_TIME	Time last LB request was sent
DSRV$L_LBMON_TIME	Time of last LB monitor pass
DSRV$L_LM_FKB	Address of load monitor thread FKB
DSRV$L_LB_FKB	Address of load balance thread FKB
DSRV$W_LM_INTERVAL	Load monitoring interval
DSRV$B_LB_COUNT1	Counter for load balancing thread
DSRV$B_LB_COUNT2	Counter for load balancing thread
DSRV$L_HULB_FL	HULB queue listhead
DSRV$L_HULB_BL	
DSRV$B_HOSTS	Bit array of hosts served
DSRV$L_UNITS	Table of UQB addresses
DSRV$L_OPCOUNT	Total count of MSCP op codes received from remote hosts. This is followed by 39 longwords which count the number of individual MSCP op codes received.
DSRV$L_ABORT_CNT	- 1 -
DSRV$L_GET_CMD_CNT	- 2 -
DSRV$L_GET_UNT_CNT	- 3 -
DSRV$L_SET_CON_CNT	- 4 -
DSRV$l_reserved	- 5 -
DSRV$l_reserved	- 6 -
DSRV$l_reserved	- 7 -
DSRV$L_AVAIL_CNT	- 8 -
DSRV$L_ONLIN_CNT	- 9 -
DSRV$L_SET_UNT_CNT	- 10 -
DSRV$L_DET_ACC_CNT	- 11 -

Field Name	Description and Flags
DSRV$l_reserved	- 12 -
DSRV$l_reserved	- 13 -
DSRV$l_reserved	- 14 -
DSRV$l_reserved	- 15 -
DSRV$L_ACCES_CNT	- 16 -
DSRV$L_CMP_CON_CNT	- 17 -
DSRV$L_ERASE_CNT	- 18 -
DSRV$L_FLUSH_CNT	- 19 -
DSRV$L_REPLC_CNT	- 20 -
DSRV$l_reserved	- 21 -
DSRV$l_reserved	- 22 -
DSRV$l_reserved	- 23 -
DSRV$l_reserved	- 24 -
DSRV$l_reserved	- 25 -
DSRV$l_reserved	- 26 -
DSRV$l_reserved	- 27 -
DSRV$l_reserved	- 28 -
DSRV$l_reserved	- 29 -
DSRV$l_reserved	- 30 -
DSRV$l_reserved	- 31 -
DSRV$L_CMP_HST_CNT	- 32 -
DSRV$L_READ_CNT	- 33 -
DSRV$L_WRITE_CNT	- 34 -
DSRV$l_reserved	- 35 -
DSRV$l_reserved	- 36 -
DSRV$l_reserved	- 37 -
DSRV$l_reserved	- 38 -
DSRV$l_reserved	- 39 -
DSRV$L_VCFAIL_CNT	Count of virtual circuit failures.
DSRV$L_BLKCOUNT	129 (decimal) counters to record the number of transfers for each transfer size handled by the server.

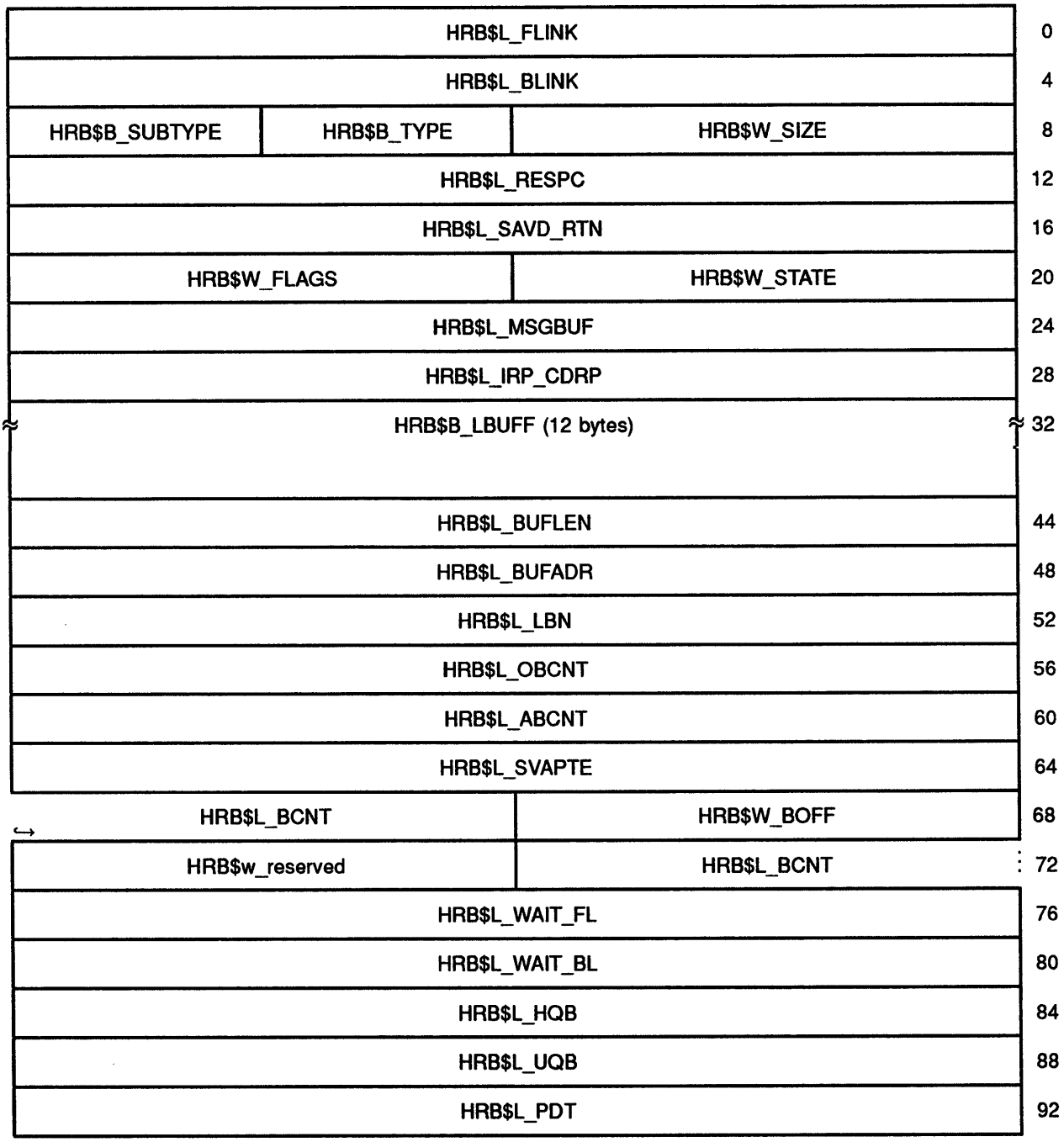
B.7 HQB - Host Queue Block



Field Name	Description and Flags
HQB$L_FLINK	Forward link to next HQB in queue of HQBs attached to DSRV.
HQB$L_BLINK	Backward link to preceding HQB in queue of HQBs attached to DSRV.
HQB$W_SIZE	Size of this data structure.
HQB$B_TYPE	Type of data structure. This field is initialized to contain the value of the symbol DYN$C_ DSRV when the HQB is allocated and initialized. This occurs when accepting a CONNECT request from a remote disk class driver.

Field Name	Description and Flags
HQB$B_SUBTYPE	Data structure subtype. This field is initialized to contain the value of the symbol DYN$C_ DSRV_HQB when the HQB is allocated and initialized. This occurs when accepting a CONNECT request from a remote disk class driver.
HQB$B_HOSTNO	Host number.
HQB$B_STATE	Host state.
The following fields are defined within HQB$B_STATE:	
HQB$V_VCFAILED	Virtual circuit to host has failed. (bit 0)
HQB$V_DISCON_INIT	Disconnect Initialization (bit 1)
HQB$V_PATHMOVE	PathMove in Progress (bit 2)
HQB$W_CNT_FLGS	Host settable controller flags.
HQB$W_HTIMO	Host access timeout interval.
HQB$W_FLAGS	Host flags.
The following fields are defined within HQB$W_FLAGS:	
HQB$V_UNIT_ONLINE	Host has a unit online. (bit 0)
HQB$V_V5CL	This is the V5 Class Driver (bit 1)
HQB$Q_TIME	Time host issued SET CONTROLLER CHARACTERISTICS.
HQB$W_NUM_QUE	Current number of outstanding requests.
HQB$W_MAX_QUE	Most requests ever outstanding.
HQB$L_HRB_FL	Forward link for head of HRB queue.
HQB$L_HRB_BL	Backward link for head of HRB queue.
HQB$L_CDT	Address of CDT for connection between local MSCP server and remote disk class driver.
HQB$B_SYSTEMID	SCS System ID of Host
HQB$L_DSRV	Address of DSRV.
HQB$L_HULB_VECTOR	HULB Vector Address
HQB$W_MAX_HULB	Size of HULB Vector

B.8 HRB - Host Request Block



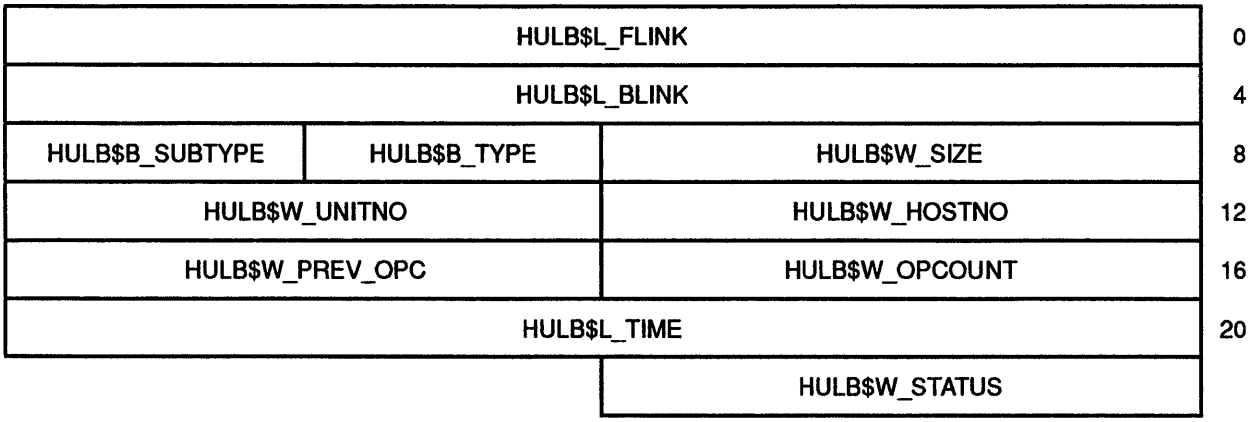
HRB$L_CMD_STS	96

Field Name	Description and Flags
HRB$L_FLINK	Forward link to next HRB in queue of HRBs attached to HQB.
HRB$L_BLINK	Backward link to preceding HRB in queue of HRBs attached to HQB.
HRB$W_SIZE	Size of this data structure.
HRB$B_TYPE	Type of data structure. This field is initialized to contain the value of the symbol DYN$C_DSRV when the HRB is allocated and initialized. This occurs when an MSCP command is received from the disk class driver on a remote host.
HRB$B_SUBTYPE	Data structure subtype. This field is initialized to contain the value of the symbol DYN$C_DSRV_ HRB when the HRB is allocated and initialized. This occurs when an MSCP command is received from the disk class driver on a remote host.
HRB$L_RESPC	PC to resume on restart.
HRB$L_SAVD_RTN	Saved address of caller.
HRB$W_STATE	State of I/O request represented by this HRB. Upon entering a "state", this field is set to contain the value of the appropriate symbol: Upon leaving a state, the HRB$V_STATE_INVALID flag is set. When this field contains 0 or has the HRB$V_STATE_INVALID flag set, the HRB is within the jurisdiction of the MSCP server.

Field Name	Description and Flags
The following values are defined within HRB$W_STATE:	
HRB$K_ST_MSG_ WAIT	Attn msg buffer/credit wait. (value 1)
HRB$K_ST_SEQ_ WAIT	Waiting for sequential cmd. (value 2)
HRB$K_ST_BUF_ WAIT	Waiting for server buffer. (value 3)
HRB$K_ST_SNDAT_ WAIT	Sending or receiving data. (value 4)
HRB$K_ST_DRV_ WAIT	Driver queue. (value 5)
HRB$K_ST_MAP_ WAIT	Mapping a buffer. (value 6)
HRB$K_ST_UNMAP_ WAIT	Returning mapping resources. (value 7)
HRB$K_ST_SNDMS_ WAIT	Sending a message. (value 8)
HRB$W_FLAGS	Status flags.
The following fields are defined within HRB$W_FLAGS:	
HRB$V_ABORT	Abort. (bit 0)
HRB$V_ABORTWS	Abort with status. (bit 1)
HRB$V_DEQUEUED	Removed from resource queues. (bit 2)
HRB$V_ENDMSG	End message needs to be sent. (bit 3)
HRB$V_MAP	Map resources allocated. (bit 4)
HRB$V_UNBLOCK	Unblock needs to be run. (bit 5)
HRB$V_VCFAILED	Virtual circuit for host failed. (bit 6)
HRB$V_OLDBUF	The buffer allocated for this request is out of the Old Buffer. (bit 7)
HRB$L_MSGBUF	Address of MSCP message buffer.
HRB$L_IRP_CDRP	Address of IRP/CDRP pair for this I/O request.
HRB$B_LBUFF	Local buffer handle. This is a 12-byte field.
HRB$L_BUFLEN	Length of buffer allocated.
HRB$L_BUFADR	Buffer starting address.
HRB$L_LBN	LBN place holder for transfer.
HRB$L_OBCNT	Original request byte count.

Field Name	Description and Flags
HRB$L_ABCNT	Accumulated byte count. The number of bytes already exchanged between the disk and the remote host.
HRB$L_SVAPTE	System virtual address of PTE for local buffer.
HRB$W_BOFF	Byte offset into first page of buffer.
HRB$L_BCNT	Temporary storage for current transfer.
HRB$W_RESERVED	Reserved
HRB$L_WAIT_FL	Forward link for HRB wait queue attached to UQB.
HRB$L_WAIT_BL	Backward link for HRB wait queue attached to UQB.
HRB$L_HQB	Address of Host Queue Block.
HRB$L_UQB	Address of Unit Queue Block.
HRB$L_PDT	Address of Port Descriptor Table.
HRB$L_CMD_STS	MSCP command status.

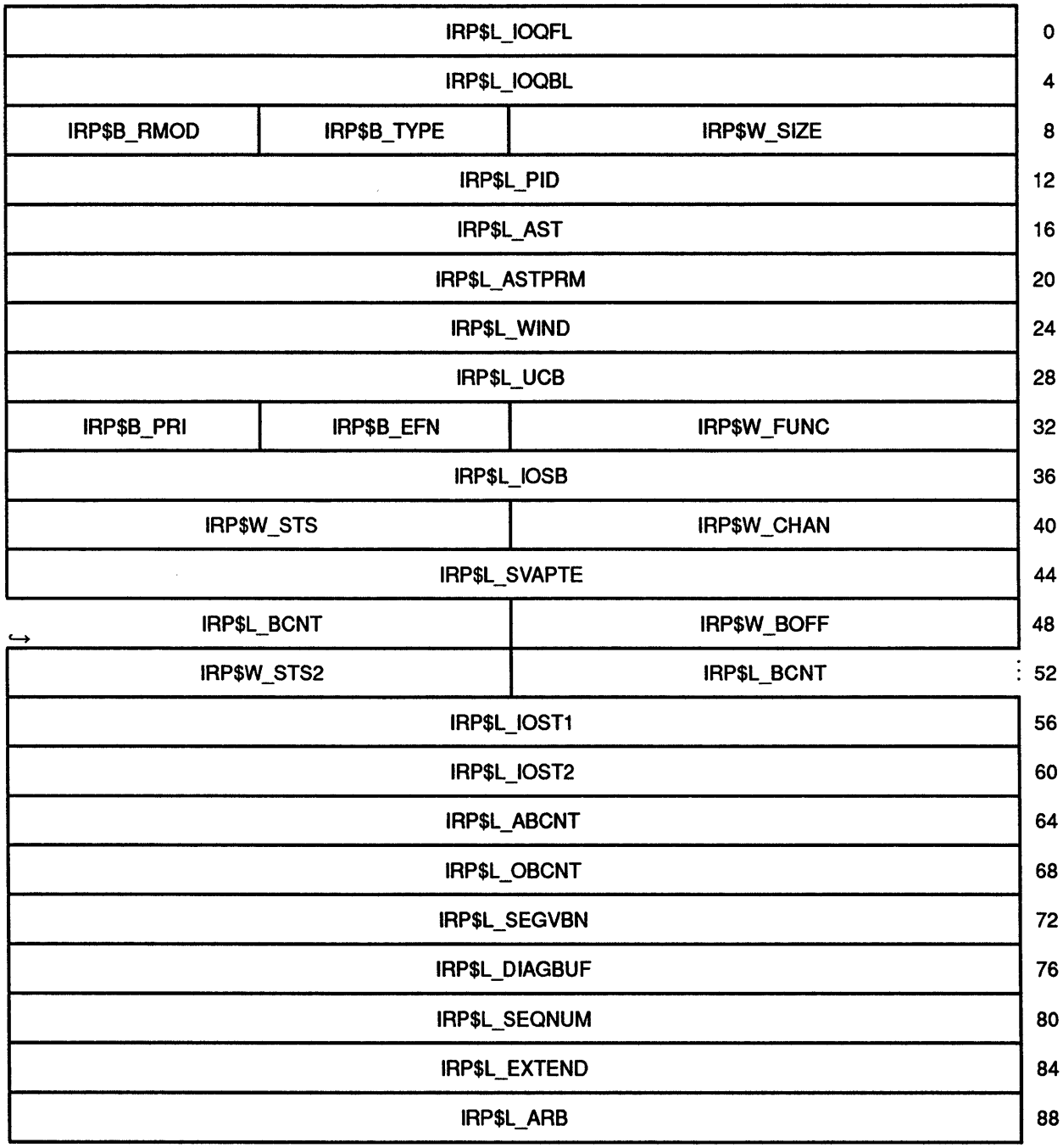
B.9 HULB - Host Unit Load Block



Field	Use
HULB$L_FLINK	Used to link this request
HULB$L_BLINK	into the DSRV data styructure
HULB$W_SIZE	Data structure size in bytes
HULB$B_TYPE	This is an MSCP type struct
HULB$B_SUBTYPE	with a HULB subtype (4)
HULB$W_HOSTNO	Assigned host number
HULB$W_UNITNO	Assigned unit number
HULB$W_OPCOUNT	Current operation count
HULB$W_PREV_OPC	Operation count for prev interval
HULB$L_TIME	Time of last Load Balance request
HULB$W_STATUS	Load Balance status bits

The following fields are defined within HULB$W_STATUS:	
HULB$V_LB_REQ	This unit has been asked to LB (bit 0)
HULB$V_DELETE	This unit is offline and the HULB can be deleted (bit 1)
HULB$V_LB_DISABLED	This unit is not available for load balancing (bit 2)
HULB$V_fill_2	Filler

B.10 IRP - I/O Request Packet



IRP$L_KEYDESC				92
IRP$L_FQFL				96
IRP$L_FQBL				100
IRP$B_FLCK	IRP$B_CD_TYPE	IRP$W_CDRPSIZE		104
IRP$L_FPC				108
IRP$L_FR3				112
IRP$L_FR4				116
IRP$L_SAVD_RTN				120
IRP$L_MSG_BUF				124
IRP$L_RSPID				128
IRP$L_CDT				132
IRP$L_RWCPTR				136
IRP$L_SHD_PIO_LNK				140
IRP$B_SHD_PIO_ERRCNT	IRP$B_SHD_PIO_FLAGS	IRP$B_SHD_PIO_ACT	IRP$B_SHD_PIO_CNT	144
IRP$L_SHD_PIO_ERROR				148
IRP$W_SHD_FILLER		$B_SHD_PIO_ERRSEV	$B_SHD_PIO_ERRINDEX	152
IRP$L_SHD_LOCK_FPC				156
IRP$L_SHD_LOCK_FR1				160
IRP$L_SHD_LOCK_FR2				164

Field Name	Description and Flags
IRP$L_IOQFL	I/O queue forward link. If I/O for a unit is stalled when this IRP is handed to DUDRIVER, the disk class driver queues the IRP to the UCB using this queue linkage. This field is the same as the IOQFL field in the CDRP since IRP fields are actually at negative offsets relative to the beginning of the associated CDRP.
IRP$L_IOQBL	I/O queue backward link.

Field Name	Description and Flags
IRP$W_SIZE	Size of this data structure.
IRP$B_TYPE	Type of data structure. This field is loaded with the value of the symbol DYN$C_IRP when the IRP is created.
IRP$B_RMOD	Access mode of the process at the time the $QIO was requested.
IRP$L_PID	PID of the process that issued the $QIO.
IRP$L_AST	Address of AST routine specified by $QIO parameter ASTADR.
IRP$L_ASTPRM	Value of $QIO parameter ASTPRM.
IRP$L_WIND	Address of Window Control Block providing mapping information for the file being accessed by the $QIO request represented by this IRP.
IRP$L_UCB	Address of UCB corresponding to the device on which the file resides.
IRP$W_FUNC	I/O function code specified by $QIO parameter FUNC.
IRP$B_EFN	Event flag number specified by $QIO parameter EFN. If the $QIO request does not specify an event flag, then event flag 0 is used by default.
IRP$B_PRI	Base priority of the process which issued the $QIO.
IRP$L_IOSB	Address of I/O status block specified by $QIO parameter IOSB.
IRP$W_CHAN	I/O channel number specified by $QIO parameter CHAN.
IRP$W_STS	Status of I/O request represented by this IRP.

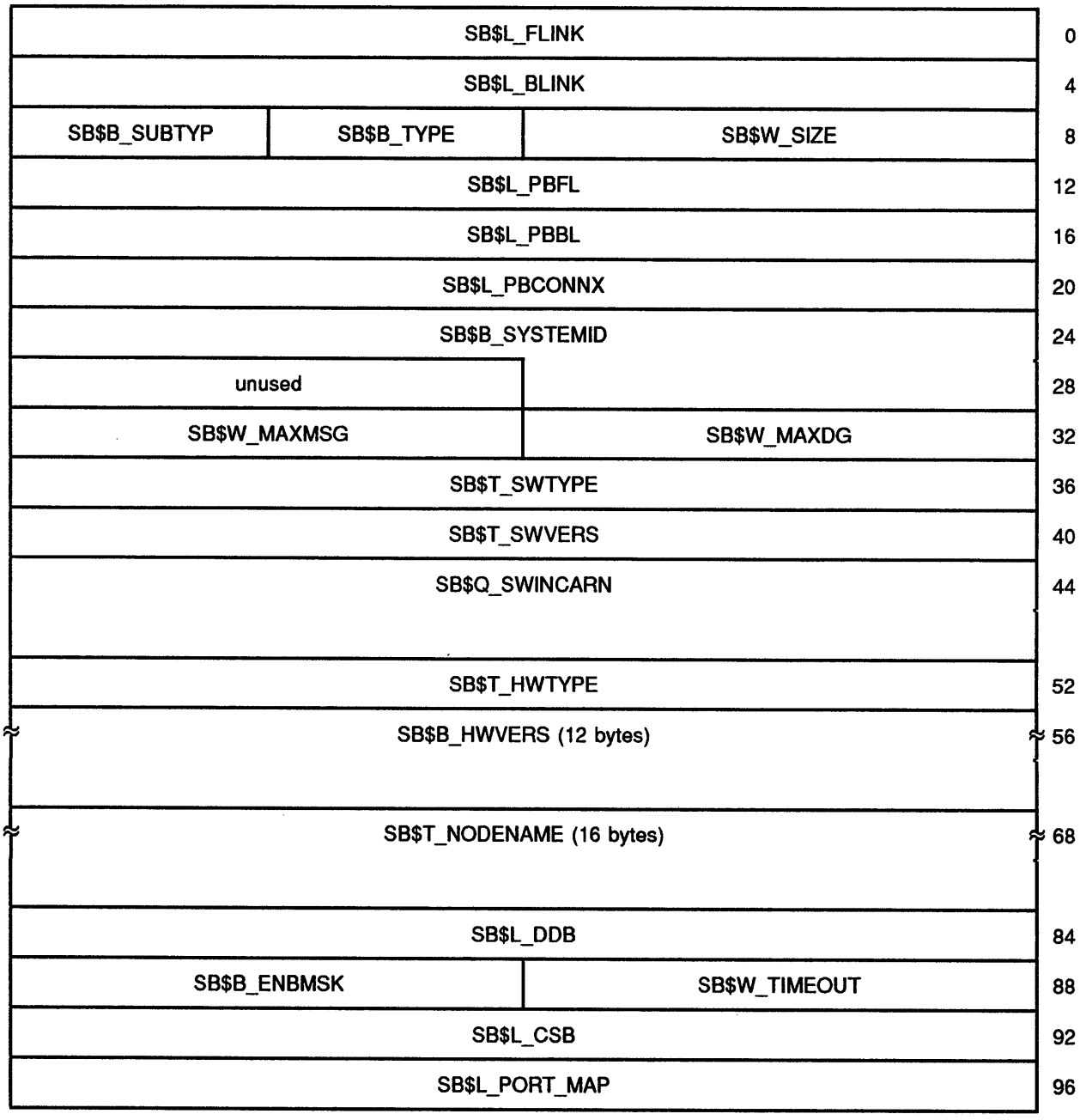
Field Name	Description and Flags
The following fields are defined within IRP$W_STS:	
IRP$V_BUFIO	Buffered I/O function. (bit 0)
IRP$V_FUNC	Read function. (bit 1)
IRP$V_PAGIO	Paging I/O function. (bit 2)
IRP$V_COMPLX	Complex-buffered-I/O function. (bit 3)
IRP$V_VIRTUAL	Virtual I/O function. (bit 4)
IRP$V_CHAINED	Chained-buffered-I/O function. (bit 5)
IRP$V_SWAPIO	Swapping I/O function. (bit 6)
IRP$V_DIAGBUF	Diagnostic buffer present. (bit 7)
IRP$V_PHYSIO	Physical I/O function. (bit 8)
IRP$V_TERMIO	Terminal I/O function. (bit 9)
IRP$V_MBXIO	Mailbox I/O function. (bit 10 (A hex))
IRP$V_EXTEND	IRP extension linked to this IRP. (bit 11 (B hex))
IRP$V_FILACP	File ACP I/O. (bit 12 (C hex))
IRP$V_MVIRP	Mount verification I/O function. (bit 13 (D hex))
IRP$V_SRVIO	SERVER TYPE I/O (TRIGGER MOUNTVER ON ERROR BUT DON'T STALL) (bit 14 (E hex))
IRP$V_KEY	KEYDESC field in use. (bit 15 (F hex))
IRP$L_SVAPTE	For a direct I/O transfer, such as a disk read or write, this field contains the address of the system PTE for the first page to be used in the transfer. This address is stored here by the routine used to lock the buffer pages in physical memory during FDT processing. For a buffered I/O transfer, the address of the buffer in system space is written here by the FDT routine which allocates the buffer. If the transfer is segmented, then this field actually applies only to the current transfer segment.
IRP$W_BOFF	For a direct I/O transfer, byte offset into first page of transfer buffer. For a buffered I/O transfer, number of bytes charged to process for transfer. FDT processing provides the information stored in this field.

Field Name	Description and Flags
IRP$L_BCNT	Count of bytes in I/O transfer. The general remarks regarding the SVAPTE field apply here as well. If the transfer is segmented, then the IRP actually represents only one segment of the transfer. This field then represents only the number of bytes in the current segment. It is updated with each new segment, and depends on how much of the total request is mapped by that segment.
IRP$W_STS2	Extension of Status Word
The following fields are defined within IRP$W_STS2:	
IRP$V_START_ PAST_HWM	I/O STARTS PAST HIGHWATER MARK (bit 0)
IRP$V_END_PAST_ HWM	I/O ENDS PAST HIGHWATER MARK (bit 1)
IRP$V_ERASE	ERASE I/O FUNCTION (bit 2)
IRP$V_PART_HWM	PARTIAL HIGHWATER MARK UPDATE (bit 3)
IRP$V_LCKIO	Locked I/O request (DECnet) (bit 4)
IRP$V_SHDIO	This is a shadowing IRP (bit 5)
IRP$V_CACHEIO	uses VBN cache buffers (bit 6)
IRP$L_IOST1	First longword to be written to I/O status block specified by $QIO parameter IOSB. An alternate name for this field is IRP$L_MEDIA.
IRP$L_IOST2	Second longword to be written to I/O status block specified by $QIO parameter IOSB.
IRP$L_ABCNT	Accumulate byte counts transferred in a $QIO request. This field is initialized to 0. Then as each segment of a transfer completes, the number of bytes transferred by that segment is added into this field.
IRP$L_OBCNT	Original transfer byte count. This field reflects the value of the $QIO parameter P2 for read and write requests. The content of the ABCNT field approaches the content of this field as segments of the request complete.
IRP$L_SEGVBN	Starting virtual block number of current transfer segment. This field is updated with each new segment.
IRP$L_DIAGBUF	Address of diagnostic buffer in system address space, if one is involved with the $QIO represented by this IRP.

Field Name	Description and Flags
IRP$L_SEQNUM	I/O transaction sequence number. If an error is logged for the $QIO request, then this field contains the universal error log sequence number.
IRP$L_EXTEND	Address of IRP extension linked to this IRP, if there is one.
IRP$L_ARB	Address of access rights block (ARB).
IRP$L_KEYDESC	Address of encryption key.
	Standard IRP must contain space for Class Driver CDRP fields.
IRP$L_FQFL	Fork Queue FLINK
IRP$L_FQBL	Fork Queue Blink
IRP$W_CDRPSIZE	Size field for positive section only
IRP$B_CD_TYPE	Type, always of interest
IRP$B_FLCK	FORK LOCK NUMBER
IRP$L_FPC	Fork PC
IRP$L_FR3	Fork R3
IRP$L_FR4	Fork R4
IRP$L_SAVD_RTN	Saved return address from level 1 JSB
IRP$L_MSG_BUF	Address of allocated MSCP buffer
IRP$L_RSPID	Allocated Request ID
IRP$L_CDT	Address of Connection Descriptor Table
IRP$L_RWCPTR	RWAITCNT pointer
	Extensions to the CDRP within the IRP Host-Based Shadowing Extension
IRP$L_SHD_PIO_LNK	Link to clone IRP(s)
IRP$B_SHD_PIO_CNT	Tot num phys IRPs assoc.
IRP$B_SHD_PIO_ACT	Tot num phys IRPs active.
IRP$B_SHD_PIO_FLAGS	Master Flags Byte
IRP$B_SHD_PIO_ERRCNT	Number of errors in chain
IRP$L_SHD_PIO_ERROR	BCNT and Error Status (SS$_)
IRP$B_SHD_PIO_ERRINDEX	Index of erring device
IRP$B_SHD_PIO_ERRSEV	Relative error severity
IRP$W_SHD_FILLER	

Field Name	Description and Flags
IRP$L_SHD_LOCK_FPC	Lock fork PC
IRP$L_SHD_LOCK_FR1	Lock fork R1
IRP$L_SHD_LOCK_FR2	Lock fork R2

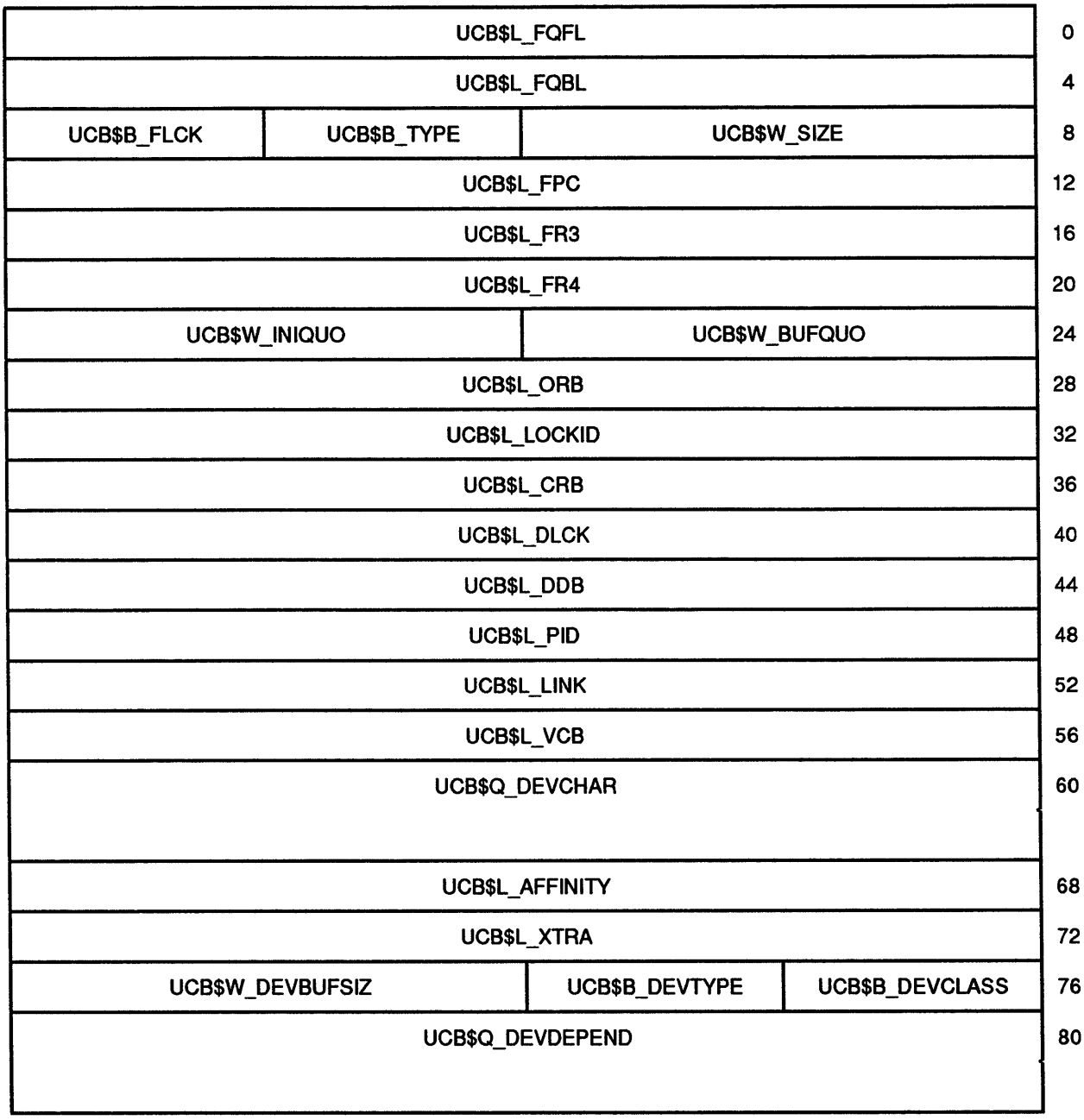
B.11 SB - System Block

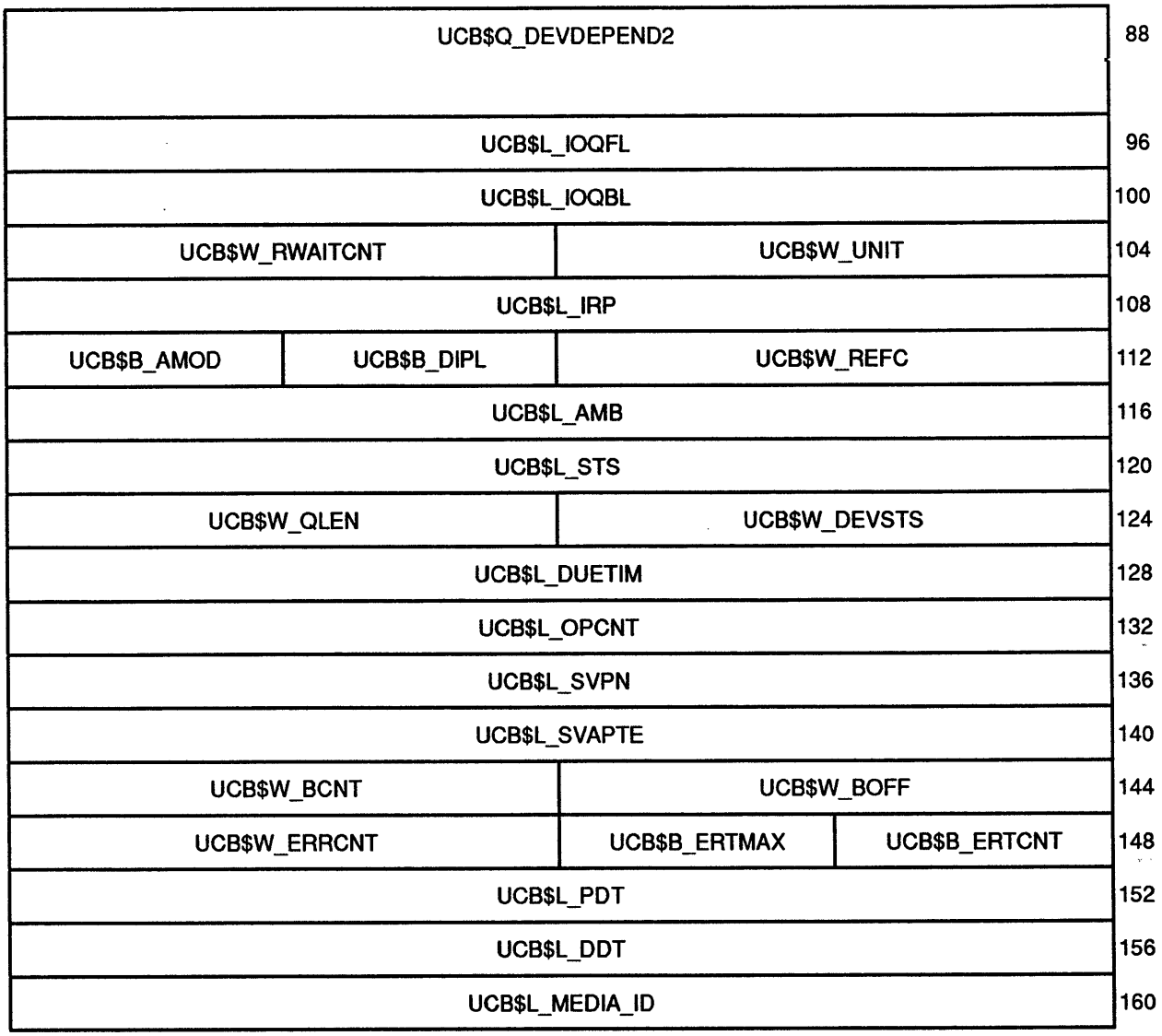


Field Name	Description and Flags
SB$L_FLINK	Forward link to next SB in queue of SBs whose header is SCS$GQ_CONFIG. The local VAX host maintains an SB corresponding to every node known to be in the cluster: one for itself, one for each other VAX, and one for each HSC. However, it must also maintain an SB for each local DSA controller (UDA50, KDB50, KDM70 etc.) as well. The microcode of such a controller implements SYSAPs which are logically equivalent to the SYSAPs implemented in the software of a VAX or an HSC. For example, DUDRIVER uses SCS services to form a connection with an MSCP server called MSCP$DISK in a local KDM70. It then exchanges MSCP packets with that server in effectively the same manner as it would with a server in a remote HSC.
SB$L_BLINK	Backward link in queue of SBs whose header is SCS$GQ_CONFIG.
SB$W_SIZE	Size of this data structure.
SB$B_TYPE	Type of data structure. This field is set to contain the value of the symbol DYN$C_SCS when the SB is created, indicating that this is one of the SCS-class of data structures.
SB$B_SUBTYP	Subtype of data structure. This field is set to contain the value of the symbol DYN$C_SCS_SB when the SB is created. This indicates that this is a System Block.
SB$L_PBFL	Forward link in Path Block queue header. A PB represents an available SCS communication path between the local host and the "system" represented by the SB to which the PB is queued.
SB$L_PBBL	Backward link in Path Block queue header.
SB$L_PBCONNX	Address of next Path Block for a connection. If multiple SCS communication paths (i.e. PBs) are available between the local host and the remote node with which a connection is desired, then SYS$SCS will select PBs on a round robin basis to provide for some load balancing. This, however, is subject to the constraint that CI and DSSI paths are given preference over NI paths.
SB$B_SYSTEMID	This 48-bit field contains the SCS system ID of the "system" represented by the SB. In the case of a VAX, this would be the value of its SYSGEN parameter SCSSYSTEMID. If the "system" were an HSC, then this is the value of the HSC's ID parameter. And if the "system" is actually a local DSA controller, then PUDRIVER constructs this quantity based on the controller's TR level and CSR address. (See description of CDDB$B_SYSTEMID field for details.)
SB$W_MAXDG	Maximum datagram size.
SB$W_MAXMSG	Maximum message size.
SB$T_SWTYPE	This 4-byte field contains up to a 4-character ASCII system software type: "VMS" or "HSC". If the SB is for other than a VAX or an HSC, then all four bytes of this field are set to 0.

Field Name	Description and Flags
SB$T_SWVERS	This 4-byte field contains up to a 4-character ASCII system software version. If the SB is for other than a VAX or HSC, then all four bytes of this field are set to 0.
SB$Q_SWINCARN	Software incarnation number. This is the time the "system" represented by the SB was initialized.
SB$T_HWTYPE	This 4-byte field contains up to a 4-character ASCII hardware type (e.g. "9000", "HS70", ...). If not a VAX or an HSC, this field is set to 0.
SB$B_HWVERS	12-byte hardware version number.
SB$T_NODENAME	16-byte field containing the SCS node name as a counted ASCII string. The first byte contains the number of characters in the string. The remaining 15 bytes contain the actual characters. Since PPD START messages limit the size of the node name to 8 bytes, that constraint applies here as well. Also, if this node is running DECnet, then this name is further restricted by DECnet to being at most 6 characters. This field is left as zeros if the SB is for a "system" other than a VAX or an HSC (e.g. local controller such as a UDA50).
SB$L_DDB	Head of DDB list linked to this SB.
SB$W_TIMEOUT	SCS Process Poller timeout field. If greater than 0, then this field represents the number of seconds remaining before this "system" becomes eligible for SCS process polling. If not greater than 0, then this "system" is eligible for polling or polling is in progress.
SB$B_ENBMSK	SCS Process Poller process enable bit mask. Indicates which processes on the "system" represented by this SB for which polling is enabled.
SB$L_CSB	Address of Cluster System Block for this "system", if one exists. CSBs are maintained only for "active" nodes, i.e. for VAXes.
SB$L_PORT_MAP	Load Sharing Port Bit Map

B.12 UCB - Unit Control Block




UCB$Q_DEVDEPEND2
88
UCB$L_IOQFL
96
UCB$L_IOQBL
100
UCB$W_RWAITCNT
UCB$W_UNIT
104
UCB$L_IRP
108
UCB$B_AMOD
UCB$B_DIPL
UCB$W_REFC
112
UCB$L_AMB
116
UCB$L_STS
120
UCB$W_QLEN
UCB$W_DEVSTS
124
UCB$L_DUETIM
128
UCB$L_OPCNT
132
UCB$L_SVPN
136
UCB$L_SVAPTE
140
UCB$W_BCNT
UCB$W_BOFF
144
UCB$W_ERRCNT
UCB$B_ERTMAX
UCB$B_ERTCNT
148
UCB$L_PDT
152
UCB$L_DDT
156
UCB$L_MEDIA_ID
160

Field Name	Description and Flags
UCB$L_FQFL	Fork queue forward link. This is a queue of UCBs that contain driver fork process contexts of drivers waiting to continue I/O processing. VMS resource management routines insert UCBs on this queue. An example would be when requesting UBA map registers via routine IOC$REQMAPREG in module IOSUBNPAG. If map registers are not available, the driver fork context is saved in the UCB and the UCB is queued to the map register wait queue (ADPL_MRQBL, ADPL_MRQFL) in the ADP for the UBA.
UCB$L_FQBL	Fork queue backward link.
UCB$W_SIZE	Size of UCB
UCB$B_TYPE	Type of data structure. This field is set to the value of the symbol DYN$C_UCB.
UCB$B_FLCK	Fork Lock Number at which device driver usually runs. For DUDRIVER, this is SPL$C_SCS.
UCB$L_FPC	Fork process driver address. Certain resource management routines, such as IOC$REQMAPREG, suspend a driver fork process and place the UCB in a resource wait queue with linkage provided by the UCB's FQFL and FQBL fields. The PC at which the fork process is to resume is stored here in the UCB$L_FPC field. When the fork process resumes, it is resumed at the IPL stored in the UCB$B_FLCK field.
UCB$L_FR3	Saved content of R3 at the time that the driver fork process is suspended.
UCB$L_FR4	Saved content of R4 at the time that the driver fork process is suspended.
UCB$W_BUFQUO	Buffered I/O quota if UCB represents mailbox.
UCB$W_INIQUO	Initial Buffered I/O Quota for this UCB.
UCB$L_ORB	Address of ORB associated with the UCB.
UCB$L_LOCKID	ID of lock on device. Used during volume mounting and also by various volume shadowing routines.
UCB$L_CRB	Address of primary CRB associated with this UCB.
UCB$L_DLCK	Address of Device IPL SpinLock
UCB$L_DDB	Address of DDB associated with this device.
UCB$L_PID	Process identification code of process that has device allocated. See $ALLOC system service documentation.
UCB$L_LINK	Address of next UCB in primary DDB chain of UCBs.
UCB$L_VCB	Address of VCB that describes the volume mounted on this device.

Field Name	Description and Flags
UCB$L_DEVCHAR	First longword of device characteristics flags for this device. Some of these flags pertain to devices other than DSA disks.
The following values are defined within UCB$L_DEVCHAR:	
DEV$V_REC	Record-oriented device (bit 0)
DEV$V_CCL	Carriage control device (bit 1)
DEV$V_TRM	Terminal device (bit 2)
DEV$V_DIR	Directory-structured device (bit 3)
DEV$V_SDI	Single directory-structured device (bit 4)
DEV$V_SOD	Sequential block-oriented device (for example, tape) (bit 5)
DEV$V_SPL	Device is spooled (bit 6)
DEV$V_OPR	Operator device (bit 7)
DEV$V_RCT	Device contains RCT (bit 8)
DEV$V_NET	Network device (bit 13 (D hex))
DEV$V_FOD	File-oriented device (bit 14 (E hex))
DEV$V_DUA	Dual-ported device (bit 15 (F hex))
DEV$V_SHR	Shareable device (used by more than one program simultaneously) (bit 10 hex)
DEV$V_GEN	Generic device (bit 17 (11 hex))
DEV$V_AVL	Device available for use (bit 18 (12 hex))
DEV$V_MNT	Device mounted (bit 19 (13 hex))
DEV$V_MBX	Mailbox device (bit 20 (14 hex))
DEV$V_DMT	Device marked for dismounting (bit 21 (15 hex))
DEV$V_ELG	Error logging enabled (bit 22 (16 hex))
DEV$V_ALL	Device allocated (bit 23 (17 hex))
DEV$V_FOR	Device mounted as foreign (bit 24 (18 hex))
DEV$V_SWL	Device software write locked (bit 25 (19 hex))
DEV$V_IDV	Device capable of providing input (bit 26 (1A hex))
DEV$V_ODV	Device capable of providing output (bit 27 (1B hex))
DEV$V_RND	Device allowing random access (bit 28 (1C hex))
DEV$V_RTM	Real-time device (bit 29 (1D hex))
DEV$V_RCK	Read-checking enabled (bit 30 (1E hex))
DEV$V_WCK	Write-checking enabled (bit 31 (1F hex))

Field Name	Description and Flags
UCB$L_DEVCHAR2	Second longword of device characteristics flags for this device. Some of these flags pertain to devices other than DSA disks.
The following values are defined within UCB$L_DEVCHAR2:	
DEV$V_CLU	Device available cluster-wide (bit 0)
DEV$V_DET	Detached terminal (bit 1)
DEV$V_RTT	Remote-terminal UCB extension (bit 2)
DEV$V_CDP	Dual-path device with two UCBs (bit 3)
DEV$V_2P	Two paths known to device (bit 4)
DEV$V_MSCP	Device accessed using MSCP (bit 5)
DEV$V_SSM	Shadow set member (bit 6)
DEV$V_SRV	Served by MSCP server (bit 7)
DEV$V_RED	Redirected terminal (bit 8)
DEV$V_NNM	See note immediately following (bit 9)

NOTE

Routine IOC$CVT_DEVNAM in module IOSUBNPAG is called from various places to perform internal conversion of a device name and unit number of a physical device name string. If the DEV$V_NNM flag is not set, then the string to be output by this conversion is limited to the format DEV: . To obtain other forms, such as those involving an allocation class or node name preceding the device name, this flag must

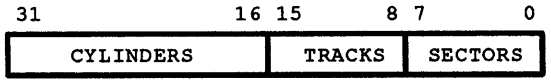
Field Name	Description and Flags
The following values are defined within UCB$L_DEVCHAR2:	
be set.	
DEV$V_WBC	Device supports write-back caching (bit 10 (A hex))
DEV$V_WTC	Device supports write-through caching (bit 11 (B hex))
DEV$V_HOC	Device supports host caching (bit 12 (C hex))
DEV$V_LOC	Device accessible via local (non-emulated) controller (bit 13 (D hex))
DEV$V_DFS	Device is DFS-served (bit 14 (E hex))
DEV$V_DAP	Device is dap accessed (bit 15 (F hex))
DEV$V_NLT	Device is not-last-track (i.e. it has no bad block information on its last track) (bit 16 (10 hex))
DEV$V_SEX	Device (tape) supports serious exception handling (bit 17 (11 hex))
DEV$V_SHD	Device is a member of a host based shadow set (bit 18 (12 hex))
DEV$V_VRT	Device is a shadow set virtual unit (bit 19 (13 hex))
DEV$V_LDR	Loader present (tapes) (bit 20 (14 hex))
DEV$V_NOLB	Device ignores server load balancing requests (bit 21 (15 hex))
DEV$V_NOCLU	Device will never be available cluster-wide (bit 22 (16 hex))
DEV$V_VMEM	Virtual member of a constituent set (bit 23 (17 hex))
UCB$L_AFFINITY	Device Affinity
UCB$L_XTRA	Extra Longword (For SMP)
UCB$B_DEVCLASS	Device class.

Field Name	Description and Flags
The following fields are defined within UCB$B_DEVCLASS:	
DC$_DISK	Disk device (value 1)
DC$_TAPE	Tape device (value 2)
DC$_SCOM	Synchronous communication device (value 32 (20 hex))
DC$_CARD	Card reader device (value 65 (41 hex))
DC$_TERM	Terminal device (value 66 (42 hex))
DC$_LP	Line Printer (value 67 (43 hex))
DC$_WORKSTATION	Workstations (value 70 (46 hex))
DC$_REALTIME	Real time device (value 96 (60 hex))
DC$_DECVOICE	DECvoice Products (value 97 (61 hex))
DC$_BUS	Buses (ie: CI) (value 128 (80 hex))
DC$_MAILBOX	Mailbox (value 160 (A0 hex))
DC$_REMCSL_ STORAGE	Remote Console Storage (value 170 (AA hex))

UCB$B_DEVTYPE	Device type. The Driver Prologue Table of every driver may specify a symbolic constant for this field. However, all such symbols are define by macro $DCDEF. The format for these symbols is DT$_xxx , where xxx is the device type. Examples would be DT$_RA60, DT$_RA82, DT$_RA90, ...
UCB$W_DEVBUFSIZ	Default buffer size. For DUDRIVER, this is initialized in the Driver Prologue Table to be 512.
UCB$L_DEVDEPEND	Contains device dependent data. For a disk device, this longword is broken down into three subfields:

```
UCB$B_SECTORS   - Sectors per track
UCB$B_TRACKS    - Tracks per cylinder
UCB$W_CYLINDERS - Cylinders per disk
```

For DSA disks, these quantities are computed by routine DU$RECORD_UNIT_STATUS in module DUDRIVER based on the contents of a GET UNIT STATUS end message.



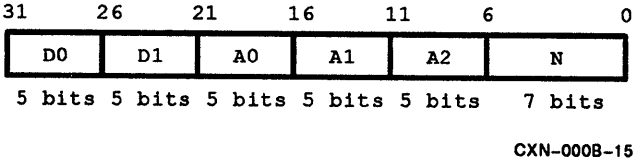
CXN-000B-14

Field Name	Description and Flags
UCB$L_DEVDEPND2	Second longword of device dependent data.
UCB$L_IOQFL	I/O queue listhead forward link. IRPs are placed in this wait queue when I/O is stalled on the device, (for example, if a disk is undergoing mount verification.) For DSA disks, DUDRIVER places IRPs in this queue when it's start I/O routine finds a nonzero RWAITCNT field in the UCB and any of the of the following conditions is true: mount verification in progress, the volume is VALID, a PACKACK is in progress, or the CDDB indicates there is no connection with the MSCP disk server in the controller. Some drivers order this queue according to base priorities of the processes which issued the IRPs in the queue. This, however, is not the case with DUDRIVER.
UCB$L_IOQBL	I/O queue listhead backward link.
UCB$W_UNIT	Number of physical device unit. For DSA disks, this field is known as the "VMS Unit Number" and is derived from the MSCPUNIT field of UCB according to the following rules: Determine if the unit is "MSCP emulated" (i.e. local to a VAX but served to the cluster by the VMS MSCP server software). The unit is MSCP emulated if the CNTRLMDL field of the CDDB is set to the value of the symbol MSCP$K_CM_EMULA (value 4). If not MSCP emulated, then the VMS unit number is a copy of the MSCPUNIT field but with the MSCP$V_SHADOW flag (high order bit) forced to 0. It does not matter here whether or not the UCB represents a shadow set virtual unit. If the unit is MSCP emulated and bits 11:8 of the MSCPUNIT field are 0, then the VMS unit number is set to the value of bits 3:0 of the MSCPUNIT field. If the unit is MSCP emulated and bits 11:8 of the MSCPUNIT field are nonzero, then the VMS unit number is set to the value of bits 7:0 of the MSCPUNIT field. Bits 11:8 of the MSCPUNIT field for an MSCP emulated disk encode the type of controller used to access the disk on the host which has direct access to it. If set to 0, then this indicates that the disk is an "old" MASSBUS or UNIBUS disk. For further details, see the functional description of routine DUTU$GET_DEVNAM in module DUTUSUBS.
UCB$W_CHARGE	Mailbox byte count quota charge, if device is a mailbox. This field and the UCB$W_RWAITCNT field are overlays of each other.

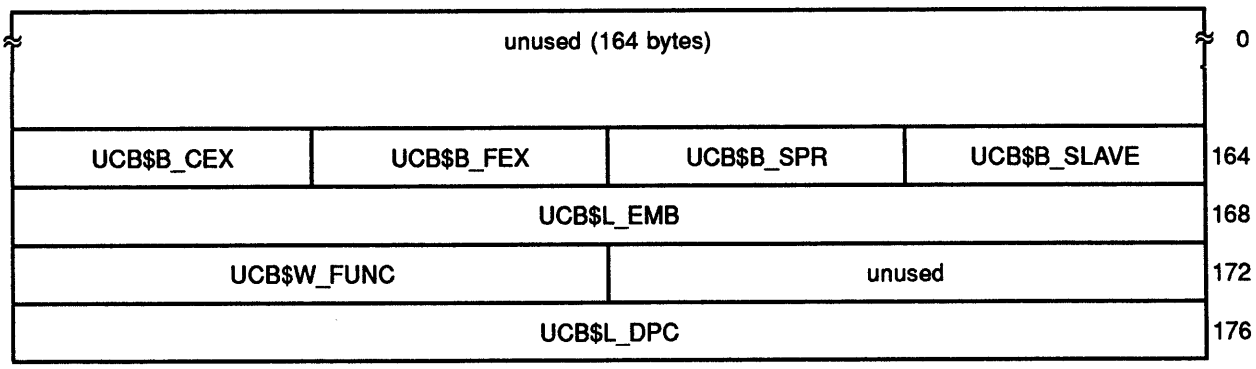
Field Name	Description and Flags
UCB$W_RWAITCNT	Resource wait count for UCB. Also nonzero if unit undergoing mount verification. This field and the UCB$W_CHARGE field are overlays of each other.
UCB$L_IRP	At the end of FDT I/O preprocessing, the address of the IRP being "queued" to the device driver is stored in this UCB field by routine IOC$INITIATE immediately before calling the driver's start I/O routine. For traditional non-DSA disks (such as MASSBUS disks), this is the address of the IRP currently being processed on the device. However, DUDRIVER can have multiple IRPs outstanding for DSA disks. Thus, this field is effectively ignored by DUDRIVER, even though IOC$INITIATE alters it each time an IRP is passed to DUDRIVER's DU_START_IO routine.
UCB$W_REFC	Reference count of processes that currently have I/O channels assigned to this device. This field is incremented by the $ASSIGN and $ALLOC system services, and decremented by the $DASSGN and $DALLOC system services.
UCB$B_DIPL	Interrupt priority level at which device requests hardware interrupts. DUDRIVER sets this field to contain IPL$_SCS. This is done since the class driver input dispatching routine, DU$IDR, is to DUDRIVER what an interrupt service routine is to a conventional device driver; and DU$IDR runs at the same IPL as the rest of DUDRIVER.
UCB$B_AMOD	Access mode at which allocation of this device occurred, if the device is allocated.
UCB$L_AMB	Associated mailbox UCB pointer. A spooled device used this field for the address of its associated device. Devices that are nonshareable and not file-oriented can use this field for the address of an associated mailbox.
UCB$L_STS	Device unit status. Some of these flags pertain to devices other than DSA disks.

Field Name	Description and Flags
The following fields are defined within UCB$L_STS:	
UCB$V_TIM	Timeout enabled (bit 0)
UCB$V_INT	Interrupts expected (bit 1)
UCB$V_ERLOGIP	Error log in progress (bit 2)
UCB$V_CANCEL	Cancel I/O on unit (bit 3)
UCB$V_ONLINE	Device is online (bit 4)
UCB$V_POWER	Power failed while unit busy (bit 5)
UCB$V_TIMOUT	Unit is timed out (bit 6)
UCB$V_INTTYPE	Receiver interrupt (bit 7)
UCB$V_BSY	Unit is busy (bit 8)
UCB$V_MOUNTING	Device is being mounted (bit 9)
UCB$V_DEADMO	Deallocated device at dismount (bit 10 (A hex))
UCB$V_VALID	Volume is software valid (bit 11 (B hex))
UCB$V_UNLOAD	Unload volume at dismount (bit 12 (C hex))
UCB$V_TEMPLATE	Template UCB from which other UCBs are made (bit 13 (D hex))
UCB$V_MNTVERIP	Mount verification is currently in progress (bit 14 (E hex))
UCB$V_WRONGVOL	Volume name does not match name in VCB (bit 15 (F hex))
UCB$V_DELETEUCB	Delete this UCB when content of REFC field becomes zero (bit 16 (10 hex))
UCB$V_LCL_VALID	Volume on this device is valid on local node (bit 17 (11 hex))
UCB$V_SUPMVMSG	Suppress mount verification messages if they indicate success (bit 18 (12 hex))
UCB$V_MNTVERPND	Mount verification pending and device busy (bit 19 (13 hex))
UCB$V_DISMOUNT	DISMOUNT IN PROGRESS (bit 20 (14 hex))
UCB$V_CLUTRAN	VAXcluster STATE TRANSITION IN PROGRESS (bit 21 (15 hex))
UCB$V_WRTLOCKMV	Write-locked mount verification in progress (bit 22 (16 hex))
UCB$V_SVPN_END	Last byte used from page mapped by SVPN (bit 23 (17 hex))
UCB$V_ALTBSY	Unit is busy via alternate startio path (bit 24 (18 hex))
UCB$V_SNAPSHOT	Restart validation is in progress (bit 25 (19 hex))
UCB$W_DEVSTS	Device-dependent status. General system flags:
The following fields are defined within UCB$W_DEVSTS:	
UCB$V_JOB	Job controller notified (bit 0)
UCB$V_TEMPL_BSY	Template UCB is busy (bit 1)

Field Name	Description and Flags
The following fields are defined within UCB$W_DEVSTS:	
	Mailbox status flags:
UCB$V_PRMMBX	Permanent mailbox (bit 0)
UCB$V_DELMBX	Mailbox marked for deletion (bit 1)
UCB$V_SHMMBX	Shared memory mailbox (bit 3) DUDRIVER (disk class driver) flags:
UCB$V_MSCP_ MNTVERIP	Mount verification is in progress (bit 8)
UCB$V_MSCP_ INITING	UCB being initialized (bit 9)
UCB$V_MSCP_ WAITBMP	RWAITCNT field has been bumped (i.e. is nonzero) (bit 10 (A hex))
UCB$V_MSCP_FLOVR	Flag is toggled each time a failover succeeds (bit 11 (B hex))
UCB$V_MSCP_PKACK	PACKACK in progress (bit 12 (C hex))
UCB$V_WRTP	Unit MSCP write protected (bit 13 (D hex))
UCB$V_MSCP_IGNSRV	Ignore Served Paths during connection Failover (bit 14 (E hex))
UCB$W_QLEN	Length of queue of IRPs whose listhead is UCB$L_IOQFL
UCB$L_DUETIM	Due time for I/O completion.
UCB$L_OPCNT	Count of operations completed on device since VMS booted. This field is modified each time an IRP is inserted into the I/O postprocessing queue.
UCB$L_SVPN	Index to virtual address of system PTE permanently allocated to device by driver loading procedure. If a Driver Prologue Table sets the DPT$M_SVP bit in the flags argument to the DPTAB macro, the driver loading procedure allocates a page of nonpaged pool to the device. Disk drivers which perform ECC correction use this page for that purpose.
UCB$L_SVAPTE	At the end of FDT I/O preprocessing, the content of the SVAPTE field in the IRP being "queued" to the device driver is stored in this UCB field by routine IOC$INITIATE immediately before calling the driver's start I/O routine. Given a traditional non-DSA disk (such as a MASSBUS disk): For a direct I/O transfer, this is the address of the system PTE for the first page to be used in the transfer. For a buffered I/O transfer, this is the address of the system buffer used in the transfer. DUDRIVER can have multiple IRPs outstanding for DSA disks. Thus, this field is effectively ignored by DUDRIVER, even though IOC$INITIATE sets it up each time it passes an IRP to DUDRIVER's DU_START_IO routine.

Field Name	Description and Flags
UCB$W_BOFF	For direct I/O transfer, byte offset in first page of transfer buffer. For buffered I/O transfer, number of bytes charged to process for transfer. The general remarks regarding the SVAPTE field apply here as well.
UCB$W_BCNT	Count of bytes in I/O transfer. The general remarks regarding the SVAPTE field apply here as well.
UCB$B_ERTCNT	Error retry count for this unit.
UCB$B_ERTMAX	Maximum error retry count allowed for a single I/O transfer.
UCB$W_ERRCNT	Number of errors that have occurred on this device since VMS booted. The DCL command SHOW DEVICE displays in its error count column the content of this field.
UCB$L_PDT	Address of Port Descriptor Table for port servicing virtual circuit to controller for this disk, if there is one.
UCB$L_DDT	Address of Driver Dispatch Table for this unit.
UCB$L_MEDIA_ID	Bit encoded media identification. This 32-bit quantity consists of five 5-bit fields and one 7-bit field. The fields are defined as follows: D0,D1 Device type name (DU, DI, DJ, DR, ...). D0 and D1 each encode one alphabetic character. "A" is encoded as a 1, "B" is encoded as a 2, *(RADHUSTVEDT)* etc. D0 encodes the left character of the device type name, and D1 encodes the right character. A0,A1,A2,N Name of media (RA60, RA90, RM80, ...) used on the unit. A0, A1, and A2 an alphabetic character or null. "A" is encoded as a 1, "B" is encoded as a 2, etc., and a 0 encodes a null (i.e. the absence of a letter). The N field encodes the two decimal digit number as a single 7-bit quantity. Thus, using decimal numbers, an RA60 would be encoded as A0 = 18 (R), A1 = 1 (A), A2 = 0 (null), and N = 60.  CXN-000B-15

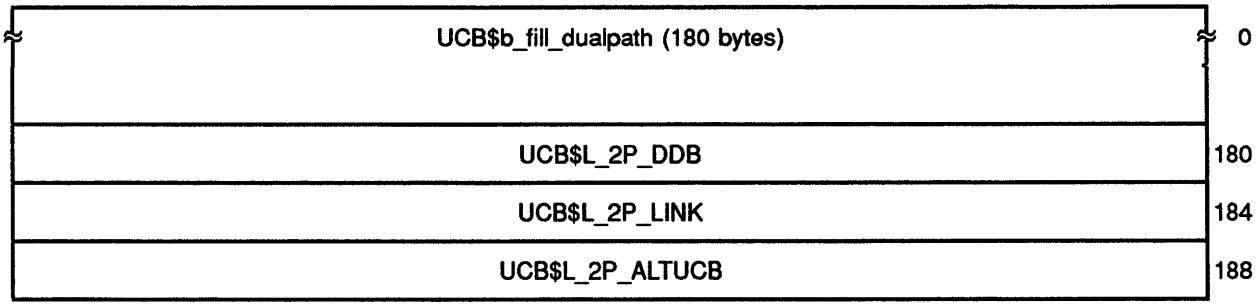
B.13 UCB Error Log Extension



Field Name	Description and Flags
UCB$B_SLAVE	Unit number of slave controller. This field is not referenced by DUDRIVER.
UCB$B_SPR	Spare byte. This field is reserved for driver use. MASSBUS adapter drivers use this field to store a fixed offset to the MASSBUS adapter registers for the unit. This field is not referenced by DUDRIVER.
UCB$B_FEX	Function dispatch table index. This field is device specific and is reserved for driver use. DUDRIVER does not reference this field.
UCB$B_CEX	Case table function execution index. This field is device specific and is reserved for driver use. DUDRIVER does not reference this field.

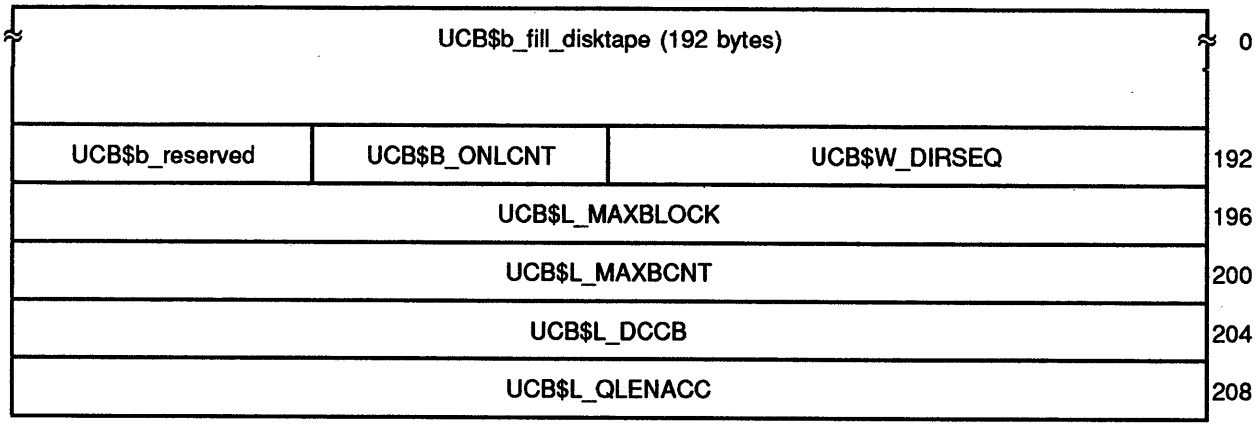
Field Name	Description and Flags
UCB$L_EMB	Address of error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate an error message buffer and copy the buffer address into this field. IOC$REQCOM writes final device status, error counters, and I/O request status into the buffer specified by this field. ERL$DEVICERR and ERL$DEVICTMO are called by drivers for older devices such as MASSBUS and UNIBUS disks. However, these routines are not referenced by DUDRIVER.
UCB$W_FUNC	I/O function modifiers. This field is read and written by various drivers that log errors. However, it is not referenced by DUDRIVER.
UCB$L_DPC	Saved driver subroutine address. This field is device specific. DUDRIVER uses this field to save the caller's return address when subroutines are called to create a new CDDB or to search a chain of DDBs for a DDB corresponding to a specific device name. See routines DUTU$CREATE_CDDB and DUTU$FIND_DDB in module DUTUSUBS.

B.14 UCB Dual Port Extension



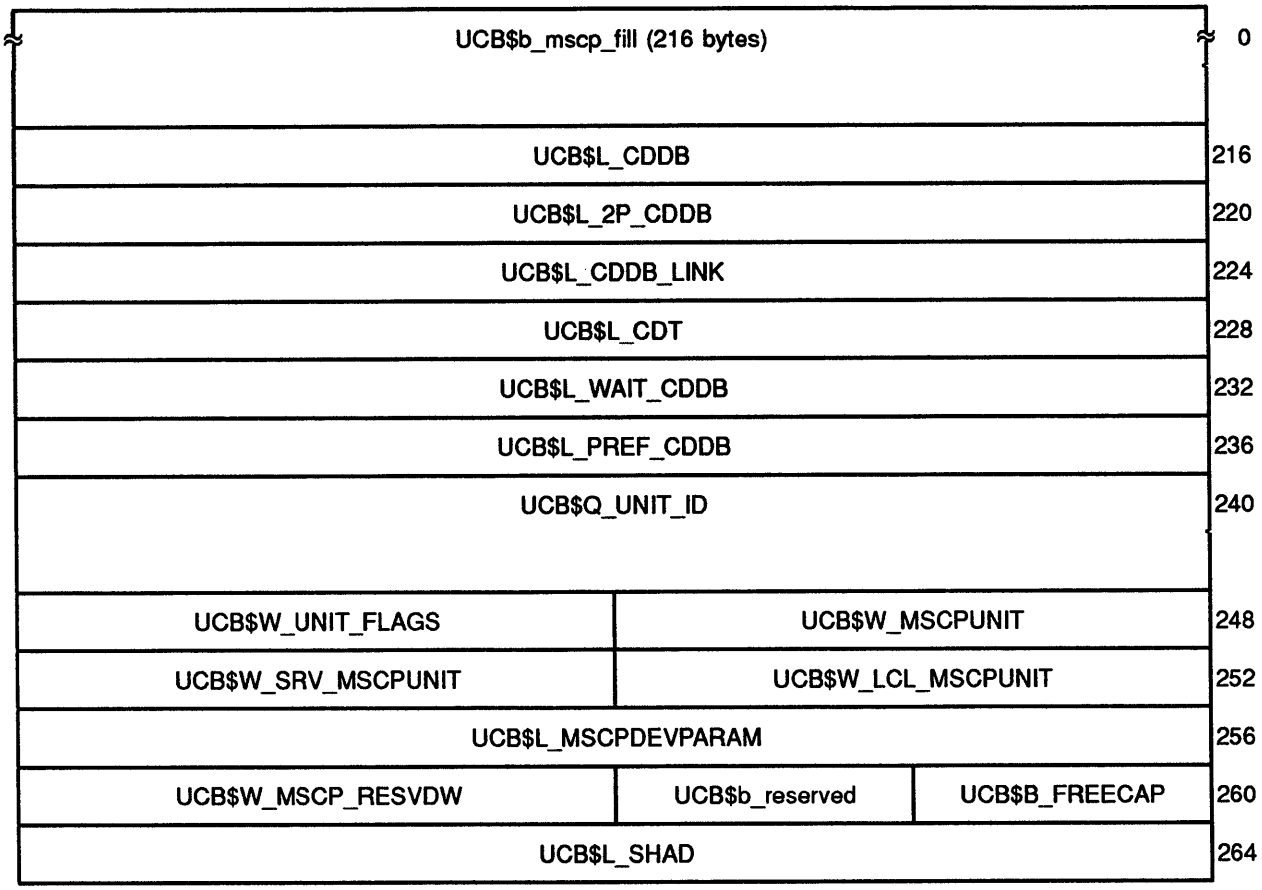
Field Name	Description and Flags
UCB$L_2P_DDB	Address of alternate DDB for secondary path.
UCB$L_2P_LINK	Address of next UCB in secondary DDB chain of UCBs.
UCB$L_2P_ALTUCB	Address of alternate UCB for this unit. This field is nonzero if the disk is dual-ported between the local host and a remote host and is also MSCP accessible via that remote host. In this case there will be two UCBs for such a disk: one representing the local path to the disk, and one representing the MSCP path to the disk via the remote host. The 2PALTUCB field in each UCB provides the address of the other UCB. An example of this would be a MASSBUS disk dual ported between two VAXes which have MSCP served the disk.

B.15 UCB Disk Extension



Field Name	Description and Flags
UCB$W_DIRSEQ	Directory sequence number. If the high order bit of this word, UCB$V_ASTARMED, is set, it indicates that the requesting process is blocking ASTs.
UCB$B_ONLCNT	Number of times unit has been placed online since VMS booted.
UCB$B_RESERVED	Reserved
UCB$L_MAXBLOCK	Maximum number of logical blocks on a random access device. DUDRIVER loads this field with a copy of the unit size field from the end message corresponding to an MSCP ONLINE command. Consequently, for DSA disks, this represents the number of logical blocks in the host area of this unit. This value does not include the logical block range occupied by the unit's Replacement and Caching Table. (The logical block number of the first block of the unit's Replacement and Caching Table is equal to this value.)
UCB$L_MAXBCNT	Maximum transfer byte count.
UCB$L_DCCB	Pointer to cache control block. DUDRIVER does not reference this field
UCB$L_QLENACC	Queue Length Accumulator

B.16 UCB MSCP Extension



Field Name	Description and Flags
UCB$L_CDDB	Address of active CDDB.
UCB$L_2P_CDDB	Address of secondary CDDB.
UCB$L_CDDB_LINK	Address of next UCB in chain of UCBs attached to active CDDB.
UCB$L_CDT	Address of CDT representing connection with MSCP$DISK in controller for this disk unit.
UCB$L_WAIT_CDDB	Address of CDDB waiting for mount verification to complete on this UCB
UCB$L_PREF_CDDB	CDDB address for Preferred Path

Field Name	Description and Flags
UCB$Q_UNIT_ID	Unique MSCP unit identifier. DUDRIVER copies into this field the unit identifier field from the end message corresponding to an MSCP ONLINE command.
UCB$W_MSCPUNIT	Primary path MSCP unit number. When a new unit is discovered, the DUDRIVER copies into this field the unit number field from the MSCP message reporting the unit. This can happen during controller initialization when polling for units is done, after controlling initialization as a result of a UNIT AVAILABLE ATTENTION message, or during creation of a shadow set. See routine DUTU$NEW_UNIT for details. **NOTE** The high order bit (i.e. bit 15 (F hex)) in this field being set indicates this UCB actually represents a shadow set virtual unit. This bit is known as the MSCP$V_SHADOW flag. Bit 14 (E hex) being set indicates that this is a Server Local Unit Number rather than a Physical Unit Number.
UCB$W_UNIT_FLAGS	MSCP unit flags
UCB$W_UNIT_FLAGS	MSCP unit flags.

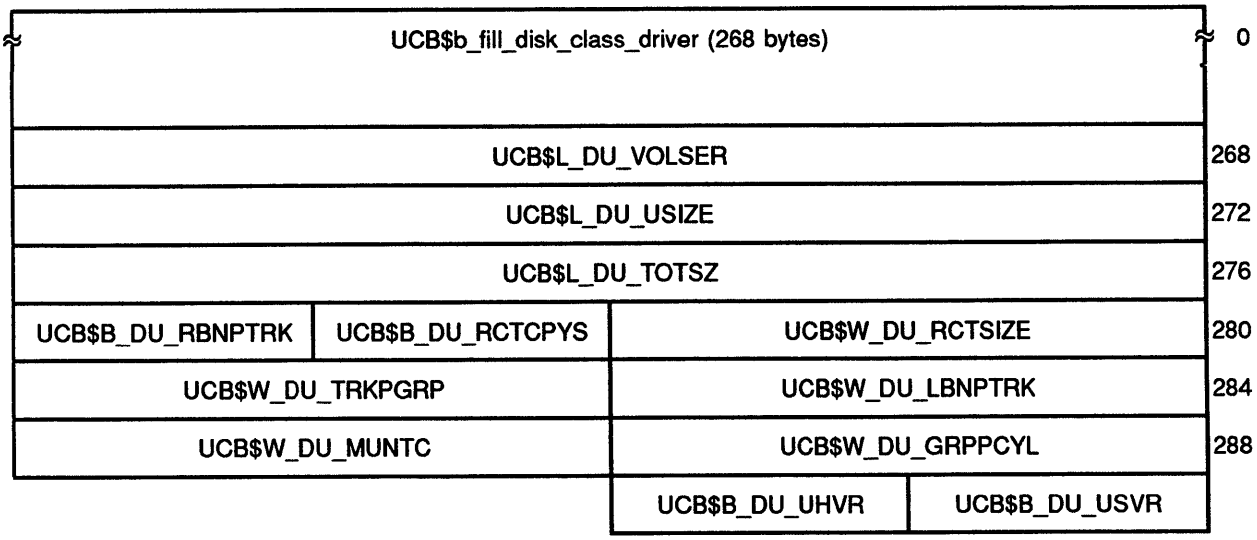
Field Name	Description and Flags
The following fields are defined within UCB$W_UNIT_FLAGS:	
MSCP$V_UF_CMPRD	Compare Reads. (bit 0) If set, all read transfers should be verified with compare operations. This characteristic is host settable, but the flag is undefined if the unit is either "unit available" or "unit offline".
MSCP$V_UF_CMPWR	Compare Writes. (bit 1) If set, all write transfers should be verified with compare operations. This characteristic is host settable, but the flag is undefined if the unit is either "unit available" or "unit offline".
MSCP$V_UF_576	576 Byte Sectors. (bit 2) The volume mounted in this unit has 576 bytes per sector. This flag is undefined if the unit is either "unit available" or "unit offline".

Field Name	Description and Flags
The following fields are defined within UCB$W_UNIT_FLAGS:	
MSCP$V_UF_WBKNV	Nonvolatile Write-Back. (bit 6) If set, all write commands should use write-back caching, rather than write-through caching, for nonvolatile caches. The controller must ensure that the existence of pending write-back data is flagged in the volume's Replacement and Caching Table before actually performing a write-back operation. This is a host settable characteristic, but the flag is undefined if the unit is either "unit available" or "unit offline".
MSCP$V_UF_RMVBL	Removable Media. (bit 7) If set, the unit has removable media. This flag is not host settable and is valid whenever the controller can determine the unit's characteristics.
MSCP$V_UF_WRTPD	Data Safety Write Protected. (bit 8) Set by the controller whenever some condition in the unit or volume prevents reliable modification of data on the volume. Possible causes include: • An incomplete bad block replacement. • An invalid Replacement Control Table (RCT). • Unit is only capable of reading volume's format (e.g. single-density volume in double-density drive).
MSCP$V_UF_SSMST	Shadow Set Master. (bit 9) If set, this UCB is represents a shadow set virtual unit. Early in the design of volume shadowing, the shadow set virtual unit was referred to as the "shadow set master unit". This nomenclature has been dropped because it implied "some form" of master/slave relationship.

Field Name	Description and Flags
The following fields are defined within UCB$W_UNIT_FLAGS:	
MSCP$V_UF_SCCHH	Suppress High Speed Caching. (bit 11 (B hex)) If set, caching using the controller's high speed cache is disabled for this unit. This is a host settable characteristic.
MSCP$V_UF_WRTPS	Software Write Protect. (bit 12 (C hex)) The host has requested software write protection for this unit.
MSCP$V_UF_WRTPH	Hardware Write Protect. (bit 13 (D hex)) The unit's hardware write protection mechanism has been activated. All write operations, including attempts to perform bad block replacement, alter the state of "Volume Write Protection", or otherwise modify the RCT, will be rejected. This flag is not valid if the unit is "unit available" or "unit offline".
MSCP$V_UF_SSMEM	Shadow Set Member. (bit 14 (E hex)) This unit is currently a member of a shadow set.
MSCP$V_UF_REPLC	Controller Initiated Bad Block Replacement (bit 15 (F hex)) If set, the controller performs bad block replacement. If clear, the does does it. This flag is undefined if the unit is "unit available" or "unit offline".
MSCP$V_UF_EXACC	Exclusive access (bit 10 (A hex))
MSCP$V_UF_CACFL	Cache flushed (bit 2)
MSCP$V_UF_EWRER	Enhanced Write Error Recovery (bit 3)
MSCP$V_UF_VARSP	Variable speed unit (bit 4)
MSCP$V_UF_VSMSU	Variable speed mode suppression (bit 5)
MSCP$V_UF_LOADR	Media Loader Present (bit 9)
MSCP$V_UF_CACH	Write-back Caching (bit 15 (F hex))
UCB$W_LCL_MSCPUNIT	MSCP unit number for local (non-emulated) controllers
UCB$W_SRV_MSCPUNIT	MSCP unit number for served (emulated) controllers
UCB$L_MSCPDEVPARAM	Device and/or controller dependent device tuning parameters. The value zero in this field means that the default or normal tuning parameters should be used. Nonzero values are intended for the selection of alternative optimization algorithms, or enabling and disabling automatic diagnosis of the unit.
UCB$B_FREECAP	Free Capacity
UCB$B_RESERVED	Reserved
UCB$W_MSCP_RESVDW	Reserved for future MSCP enhancements.

Field Name	Description and Flags
UCB$L_SHAD	Virtual Unit Pointer to HBS SHAD

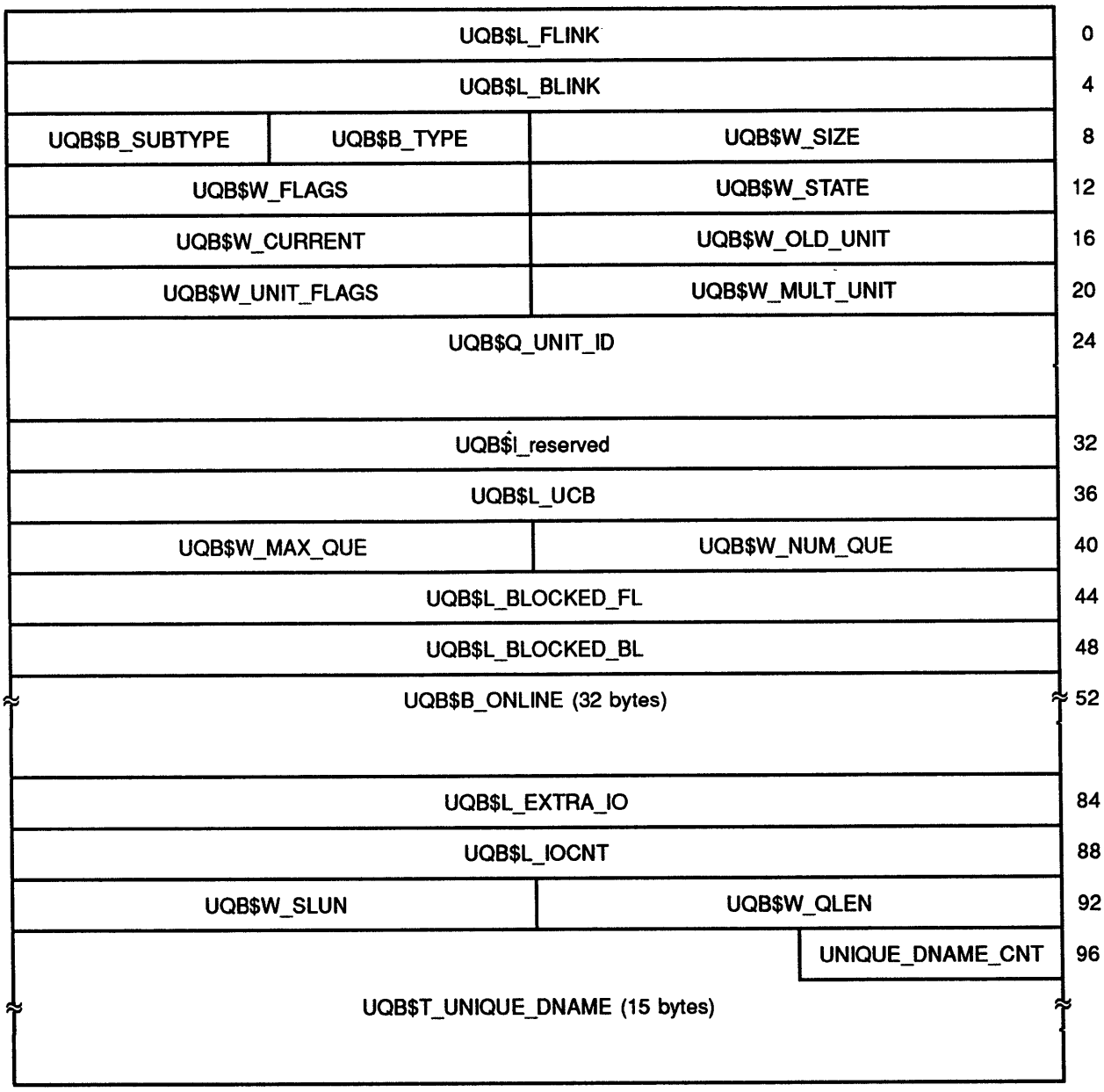
B.17 UCB DUDRIVER Extension



Field Name	Description and Flags
UCB$L_DU_VOLSER	Volume serial number as returned in ONLINE end message.
UCB$L_DU_USIZE	Size of host visible area of unit in logical blocks.
UCB$L_DU_TOTSZ	Size of unit, including RCT area, in logical blocks.
UCB$W_DU_RCTSIZE	Size of RCT area in blocks.
UCB$B_DU_RCTCPYS	Number of RCT copies on this unit.
UCB$B_DU_RBNPTRK	Number of RBNs per track.
UCB$W_DU_LBNPTRK	Number of LBNs per track. Used by volume shadowing to insure that members of a shadow set virtual unit have the same geometry.
UCB$W_DU_TRKPGRP	Number of tracks per group. Used by volume shadowing to insure that members of a shadow set virtual unit have the same geometry.
UCB$W_DU_GRPPCYL	Number of groups per cylinder. Used by volume shadowing to insure that members of a shadow set virtual unit have the same geometry.
UCB$W_DU_MUNTC	Multi-unit code. Used during host initiated BBR.

Field Name	Description and Flags
UCB$B_DU_USVR	Unit software version. Used during host initiated BBR.
UCB$B_DU_UHVR	Unit hardware version. Used during host initiated BBR.

B.18 UQB - Unit Queue Block



The following are the contents of the aggregate structure UQBDEF:

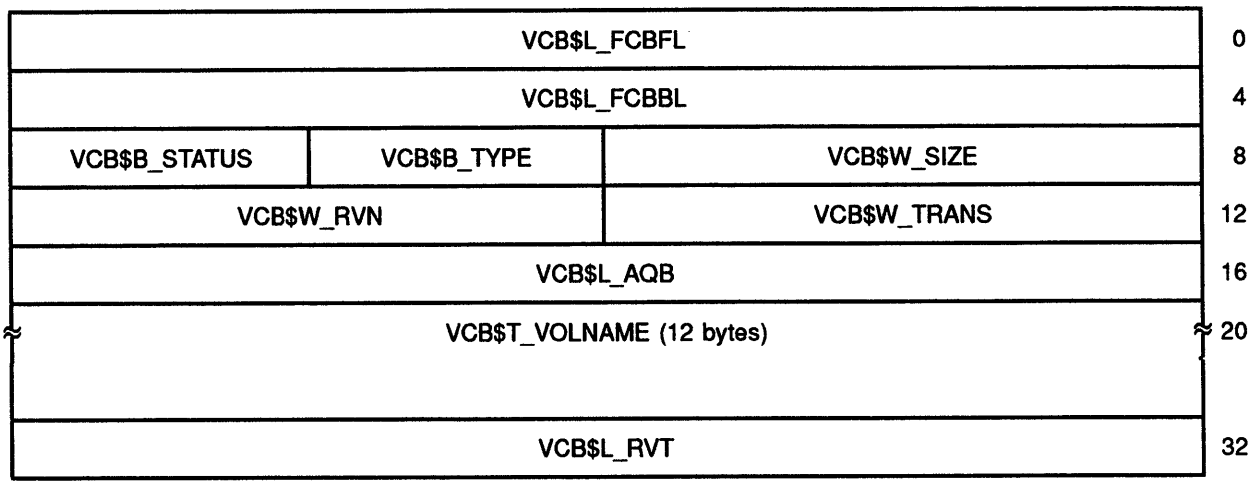
Field	Use
UQB$L_FLINK	Used to link together all
UQB$L_BLINK	UQBs being served
UQB$W_SIZE	Structure size in bytes
UQB$B_TYPE	MSCP type structure
UQB$B_SUBTYPE	with a UQB subtype (5)
UQB$W_STATE	Current state of this unit
The following values are defined for UQB$W_STATE:	
UQB$K_ST_ONLINE (value 2)	Sequential command executing
UQB$K_ST_OFFLINE (value 3)	Unit is offline
UQB$K_ST_AVAILABLE (value 4)	Unit is available
UQB$W_FLAGS	Unit usage
The following fields are defined within UQB$W_FLAGS:	
UQB$V_SEQ	This field is 1 bit long, and starts at bit 0 (0 hex). Sequential command executing
UQB$V_WRTPH	This field is 1 bit long, and starts at bit 1 (1 hex). Unit is writelocked
UQB$V_WRTPS	This field is 1 bit long, and starts at bit 2 (2 hex). Unit was mounted /NOWRITE
UQB$W_OLD_UNIT	"Old Style" unit number
UQB$W_CURRENT	Commands active on this unit
UQB$W_MULT_UNIT	This information is set up
UQB$W_UNIT_FLAGS	in ADDUNIT when the device
UQB$Q_UNIT_ID	is set /SERVED.
UQB$L_ALLOCLS	The unit identifier is made up
UQB$W_UNIT	of the allocation class, the unit and the device name
UQB$W_DEVNAME	

Field	Use
The following fields are defined within UQB$W_DEVNAME:	
UQB$V_C	This field is 5 bits long, and starts at bit 0 (0 hex). UCB unit number, the controller
UQB$V_D1	This field is 5 bits long, and starts at bit 5 (5 hex). letter, and the D1 D0 fields
UQB$V_D0	This field is 5 bits long, and starts at bit 10 (A hex). from the media ID field
UQB$l_reserved	
UQB$L_UCB	UCB address for this unit
UQB$W_NUM_QUE	Host requests pending
UQB$W_MAX_QUE	Most requests ever pending
UQB$L_BLOCKED_FL	List head for HRBs pending
UQB$L_BLOCKED_BL	sequential cmd completion
UQB$B_ONLINE	Array of hosts with unit online
UQB$L_EXTRA_IO	Splinter requests
UQB$L_IOCNT	Server contribution to total
UQB$W_QLEN	Server queue length for unit
UQB$W_SLUN	Server local unit number
UQB$B_UNIQUE_ DNAME_CNT	.ASCIC string with
UQB$T_UNIQUE_ DNAME	Cluster unique name for disk

The following constants are defined in conjunction with UQBDEF:

Constant	Value	Use
UQB$C_LENGTH	112	
UQB$K_LENGTH	112	

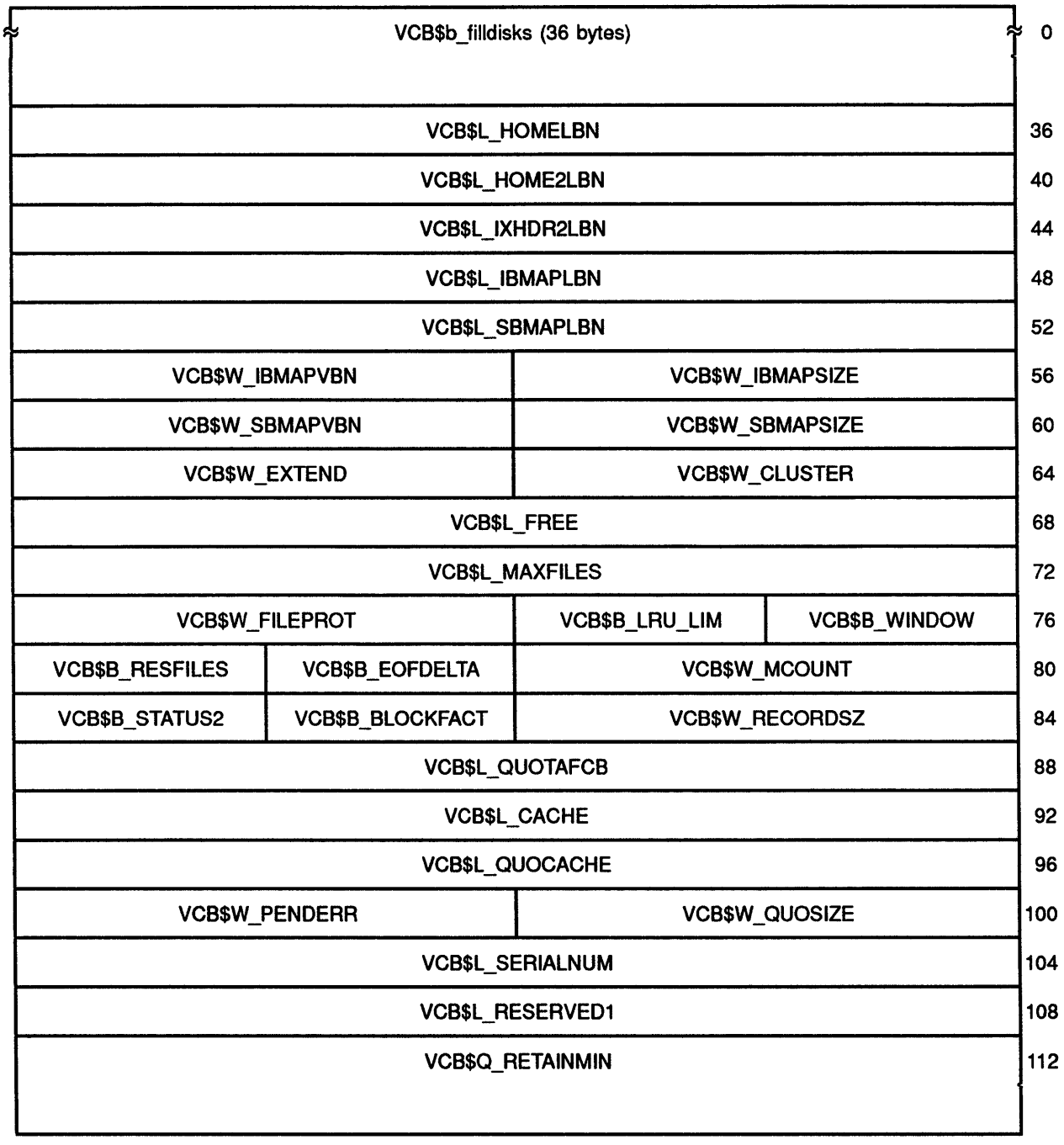
B.19 VCB - Volume Control Block Common Definitions

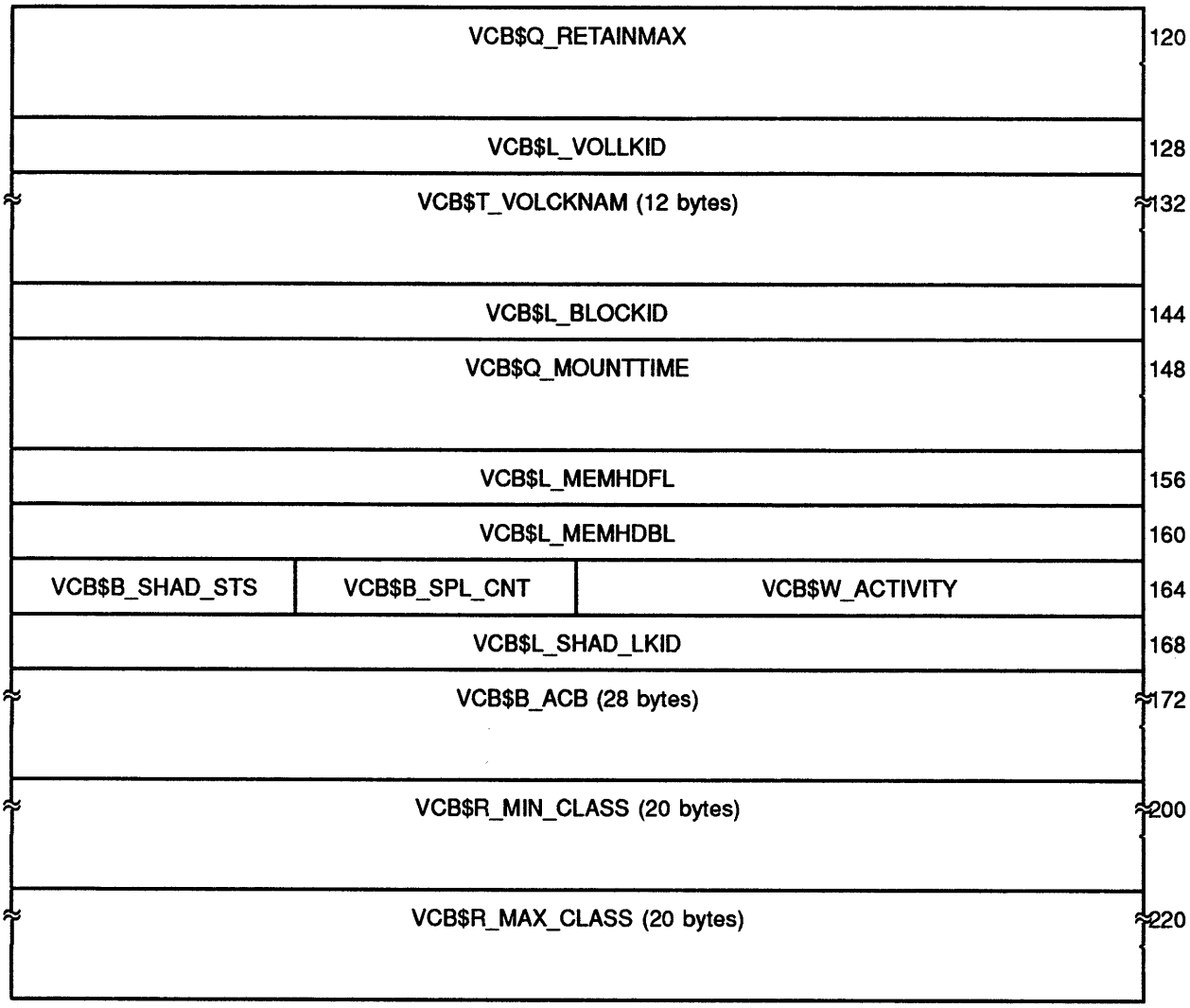


Field Name	Description and Flags
VCB$L_FCBFL	FCB listhead forward link.
VCB$L_BLOCKFL	Blocked request listhead forward link.
VCB$L_FCBBL	FCB listhead backward link.
VCB$L_BLOCKBL	Blocked request listhead backward link.
VCB$W_SIZE	Size of this data structure in bytes.
VCB$B_TYPE	Data structure type. This field is set to contain the value of the symbol DYN$C_VCB when the VCB is created.
VCB$B_STATUS	Volume status.

Field Name	Description and Flags
The following fields are defined within VCB$B_STATUS:	
VCB$V_WRITE_IF	Index file is write accessed. (bit 0)
VCB$V_WRITE_SM	Storage map is write accessed. (bit 1)
VCB$V_HOMBLKBAD	Primary homeblock is bad. (bit 2)
VCB$V_IDXHDRBAD	Primary index file header is bad. (bit 3)
VCB$V_NOALLOC	Allocation/deallocation inhibited. (bit 4)
VCB$V_EXTFID	Volume has 24 bit file numbers. (bit 5)
VCB$V_GROUP	Volume is mounted /GROUP. (bit 6)
VCB$V_SYSTEM	Volume is mounted /SYSTEM. (bit 7)
VCB$V_SHADMAST	This VCB is for shadow set master (bit 0)
VCB$V_FAILED	Member failed out of shadow set (bit 1)
VCB$V_REBLDNG	Mount verfication rebuilding shadow set (bit 3)
VCB$V_BLKASTREC	Shadowing lock blocking AST received (bit 4)
VCB$V_MVBEGUN	Mount verification initiated (bit 5)
VCB$V_ADDING	Adding member to shadow set (bit 6)
VCB$V_PACKACKED	Member PACKACKed during rebuild attempt (bit 7)
VCB$W_TRANS	Volume transaction count.
VCB$W_RVN	Relative volume number.
VCB$L_AQB	Address of AQB.
VCB$T_VOLNAME	Volume label. This field is 12 bytes in length, blank filled.
VCB$L_RVT	Address of UCB or Relative Volume Table.

B.19.1 Volume Control Block fields for Disks





Field Name	Description and Flags
VCB$L_HOMELBN	LBN of volume homeblock.
VCB$L_HOME2LBN	LBN of alternate homeblock.
VCB$L_IXHDR2LBN	LBN of alternate index file header.
VCB$L_IBMAPLBN	LBN of index file bitmap.
VCB$L_SBMAPLBN	LBN of storage bitmap.
VCB$B_IBMAPSIZE	Size of index file bitmap.
VCB$B_IBMAPVBN	Current VBN in index file bitmap.

Field Name	Description and Flags
VCB$B_SBMAPSIZE	Size of storage bitmap.
VCB$B_SBMAPVBN	Current VBN in storage bitmap.
VCB$W_CLUSTER	Volume cluster size.
VCB$W_EXTEND	Volume default file extension length.
VCB$L_FREE	Number of free blocks on volume.
VCB$L_MAXFILES	Maximum number of files allowed on volume.
VCB$B_WINDOW	Volume default window size.
VCB$B_LRU_LIM	Volume directory LRU size limit.
VCB$W_FILEPROT	Volume default file protection.
VCB$W_MCOUNT	Mount count.
VCB$B_EOFDELTA	Index file EOF update count.
VCB$B_RESFILES	Number of reserved files on volume.
VCB$W_RECORDSZ	Number of tyes in a record.
VCB$B_BLOCKFACT	Volume blocking factor.
VCB$B_STATUS2	Second status byte.
The following fields are defined within VCB$B_STATUS2:	
VCB$V_WRITETHRU	Volume is to be write-through cached. (bit 0)
VCB$V_NOCACHE	All caching is disabled on volume. (bit 1)
VCB$V_MOUNTVER	Volume can undergo mount verification. (bit 2)
VCB$V_ERASE	Erase data when blocks removed from file. (bit 3)
VCB$V_NOHIGHWATER	Turn off high-water marking. (bit 4)
VCB$V_NOSHARE	Non-shared mount. (bit 5)
VCB$V_CLUSLOCK	Cluster-wide locking necessary. (bit 6)
VCB$V_SUBSET0	ODS-2 subset 0 volume. (bit 7)
VCB$L_QUOTAFCB	Address of FCB of disk quota file.
VCB$L_CACHE	Address of volume cache block.
VCB$L_QUOCACHE	ADdress of volume quota cache.
VCB$W_QUOSIZE	Length of quota cache to allocate.
VCB$W_PENDERR	Count of pending write errors.
VCB$L_SERIALNUM	Volume serial number.
VCB$L_RESERVED1	Reserved.
VCB$Q_RETAINMIN	Minimum file retention period.
VCB$Q_RETAINMAX	Maximum file retention period.

Field Name	Description and Flags
VCB$L_VOLLKID	Volume lock ID.
VCB$T_VOLCKNAM	Volume lock name.
	This field is 12 bytes in length.
VCB$L_BLOCKID	Volume blocking lock.
VCB$Q_MOUNTTIME	Volume mount time.
VCB$L_MEMHDFL	Shadow set members queue header forward link.
VCB$L_MEMHDBL	Shadow set members queue header backward link.
VCB$W_ACTIVITY	Activity count/flag.
VCB$B_SPL_CNT	Number of Devices Spooled to Volume
VCB$B_SHAD_STS	Status byte relative to MEMHDFL.
VCB$L_SHAD_LKID	Shadowing lock lock-id.
VCB$B_ACB	ACB for blocking AST.
	This field is 28 bytes in length.
VCB$R_MIN_CLASS	Minimum classification.
	This field is 20 bytes in length.
VCB$R_MAX_CLASS	Maximum classification.
	This field is 20 bytes in length.

Appendix C

Cross Reference

This appendix provides a cross reference to find the routines mentioned in this book. The left column contains routine names, and the right column contains the modules in VMS where the routines can be found.

Table C–1: VMS Routine and Module Cross Reference

Routine	Module
ABORTS_DONE_TEST_CANCEL	[DRIVER.LIS]DUTUSUBS.LIS
ABORT	[MSCP.LIS]MSCP.LIS
ABORT_READ	[MSCP.LIS]MSCP.LIS
ABORT_UNHOOK_CDRP	[MSCP.LIS]MSCP.LIS
ABORT_WRITE	[MSCP.LIS]MSCP.LIS
ACCESS	[MSCP.LIS]MSCP.LIS
ACCESS_PATH_ATTN	[DRIVER.LIS]DUDRIVER.LIS
ACCVIO	[SYS.LIS]SYSACPFDT.LIS
ACCVIO	[SYS.LIS]SYSQIOREQ.LIS
ACP$ACCESSNET	[SYS.LIS]SYSACPFDT.LIS
ACP$ACCESS	[SYS.LIS]SYSACPFDT.LIS
ACP$DEACCESS	[SYS.LIS]SYSACPFDT.LIS
ACP$MODIFY	[SYS.LIS]SYSACPFDT.LIS
ACP$MOUNT	[SYS.LIS]SYSACPFDT.LIS
ACP$READBLK	[SYS.LIS]SYSACPFDT.LIS
ACP$WRITEBLK	[SYS.LIS]SYSACPFDT.LIS
ACTIVE	[MSCP.LIS]MSCP.LIS
ADD	[MSCP.LIS]MSCP.LIS
ADD_DOLLAR	[SYS.LIS]IOSUBNPAG.LIS
ADD_NODE	[SYS.LIS]IOSUBNPAG.LIS

Table C–1 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
AFTER_DEBUGGING_SANITY_CHECK	[SYS.LIS]IOSUBNPAG.LIS
AFTER_MAP	[DRIVER.LIS]DUDRIVER.LIS
ALLOCATE	[MSCP.LIS]MSCP.LIS
ALLOCATE_HRB	[MSCP.LIS]MSCP.LIS
ALLOCATE_HULB	[MSCP.LIS]MSCP.LIS
ALLOC	[SYS.LIS]SYSQIOREQ.LIS
ALLOC_BD	[DRIVER.LIS]SCSXPORT.LIS
ALLOC_DESCRIP	[SYS.LIS]IOSUBNPAG.LIS
ALLOC_NAME	[SYS.LIS]IOSUBNPAG.LIS
AST_REC	[BOOTS.LIS]CONFIGMN.LIS
ATTN_MSG	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE	[MSCP.LIS]MSCP.LIS
AVAILABLE_ABORT	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_CTRLERR	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_DRVERR	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_MEDOFL	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_SSSC	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_SUCC	[DRIVER.LIS]DUDRIVER.LIS
AVAILABLE_THRUPUT_PORTS	[SYSLOA.LIS]SYS$SCS.LIS
AVAIL_CDTERR	[DRIVER.LIS]DUDRIVER.LIS
AVAIL_IVCMD	[DRIVER.LIS]DUDRIVER.LIS
AVAIL_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
BACK	[MSCP.LIS]MSCP.LIS
BAD_CONID	[DRIVER.LIS]SCSXPORT.LIS
BAD_CONN	[MSCP.LIS]MSCP.LIS
BAD_FLAGS	[MSCP.LIS]MSCP.LIS
BAD_LCONID	[DRIVER.LIS]SCSXPORT.LIS
BAD_LEN	[MSCP.LIS]MSCP.LIS
BAD_MOD	[MSCP.LIS]MSCP.LIS
BAD_OPC	[MSCP.LIS]MSCP.LIS
BAD_RSPID	[DRIVER.LIS]DUDRIVER.LIS
BAD_SCONID	[DRIVER.LIS]SCSXPORT.LIS
BAD_UNIT	[MSCP.LIS]MSCP.LIS
BD_SEQ_ERROR	[DRIVER.LIS]SCSXPORT.LIS
BGXERR	[DRIVER.LIS]DUTUSUBS.LIS

Table C–1 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
BLOCKED	[MSCP.LIS]MSCP.LIS
BRACCVIO1	[SYS.LIS]SYSACPFDT.LIS
BRACCVIO	[SYS.LIS]SYSACPFDT.LIS
BRMODIFY	[SYS.LIS]SYSACPFDT.LIS
BRXQUOTA	[SYS.LIS]SYSACPFDT.LIS
BUILDACPBUF	[SYS.LIS]SYSACPFDT.LIS
CALCAVG11	[SYSLOA.LIS]SYS$SCS.LIS
CALCULATE_LOAD	[SYSLOA.LIS]SYS$SCS.LIS
CALC_AVAIL	[SYSLOA.LIS]SYS$SCS.LIS
CALC_CRC	[SYSLOA.LIS]MOUNTVER.LIS
CALC_EB	[SYSLOA.LIS]SYS$SCS.LIS
CALC_MAX_MIN	[SYSLOA.LIS]SYS$SCS.LIS
CALC_MVTIMEOUT	[SYSLOA.LIS]MOUNTVER.LIS
CALC_PORT_LOAD	[SYSLOA.LIS]SYS$SCS.LIS
CALL_DRIVER_MNTVER	[SYSLOA.LIS]MOUNTVER.LIS
CAL_DOMAIN	[SYSLOA.LIS]SYS$SCS.LIS
CANT_MV	[SYSLOA.LIS]MOUNTVER.LIS
CDDB_INIT_PRM_CDRP	[DRIVER.LIS]DUTUSUBS.LIS
CDTFOUND	[SYSLOA.LIS]SYS$SCS.LIS
CHECKSUM_BUF	[SYSLOA.LIS]MOUNTVER.LIS
CHECK_CURRENT	[MSCP.LIS]MSCP.LIS
CHECK_DDB_CHAIN	[SYS.LIS]IOSUBNPAG.LIS
CHECK_DISK	[MSCP.LIS]MSCP.LIS
CHECK_MIN_PORT_WAIT	[SYSLOA.LIS]SYS$SCS.LIS
CHECK_SERVICE	[MSCP.LIS]MSCP.LIS
CHECK_SYSTEM_DISK	[SYSLOA.LIS]MOUNTVER.LIS
CHECK_SYSTEM_QUORUM_DISK	[SYSLOA.LIS]MOUNTVER.LIS
CHKDESCR	[SYS.LIS]SYSACPFDT.LIS
CHKDISMOUNT	[SYS.LIS]SYSACPFDT.LIS
CHKDON	[SYS.LIS]SYSQIOREQ.LIS
CHKMOUNT	[SYS.LIS]SYSACPFDT.LIS
CHKRED	[SYSLOA.LIS]SYS$SCS.LIS
CHK_CRWAIT	[DRIVER.LIS]SCSXPORT.LIS
CI780_EB	[SYSLOA.LIS]SYS$SCS.LIS
CIBCAA_EB	[SYSLOA.LIS]SYS$SCS.LIS

Table C–1 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
CIBCAB_EB	[SYSLOA.LIS]SYS$SCS.LIS
CIBCI_EB	[SYSLOA.LIS]SYS$SCS.LIS
CIXCD_EB	[SYSLOA.LIS]SYS$SCS.LIS
CK_FOR	[SYSLOA.LIS]MOUNTVER.LIS
CK_STS	[SYSLOA.LIS]MOUNTVER.LIS
CLEANUP_HRB	[MSCP.LIS]MSCP.LIS
CLEANUP_IO	[SYSLOA.LIS]MOUNTVER.LIS
CLEANUP_RTN	[SYSLOA.LIS]SYS$SCS.LIS
CLREF	[SYS.LIS]SYSQIOREQ.LIS
CLR_R1_EXIT	[DRIVER.LIS]DUDRIVER.LIS
COMMON_ALOUBAMAP	[SYS.LIS]IOSUBNPAG.LIS
COMMON_CALC	[SYSLOA.LIS]SYS$SCS.LIS
COMMON_EXIT	[SYSLOA.LIS]MOUNTVER.LIS
COMMON_PHYS_IO	[DRIVER.LIS]DUDRIVER.LIS
COMMON_XFER	[DRIVER.LIS]SCSXPORT.LIS
COMPARE_HOST_DATA	[MSCP.LIS]MSCP.LIS
COMP_CTRL_DATA	[MSCP.LIS]MSCP.LIS
CONNECTION_MOVE_COUNT	[SYSLOA.LIS]SYS$SCS.LIS
CONTINUE_LOOP	[SYS.LIS]IOSUBNPAG.LIS
COPY_ASCIC	[SYSLOA.LIS]MOUNTVER.LIS
COPY_CHAR	[MSCP.LIS]MSCP.LIS
COPY_STRING	[SYSLOA.LIS]MOUNTVER.LIS
CRC_TABLE	[SYSLOA.LIS]MOUNTVER.LIS
CREATE_LOG	[MSCP.LIS]MSCP.LIS
CREATE_UQB	[MSCP.LIS]MSCP.LIS

Table C–2: VMS Routine and Module Cross Reference

Routine	Module
DACSPND	[SYS.LIS]SYSQIOREQ.LIS
DBL_WAIT_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DDB_VALIDATED	[SYS.LIS]IOSUBNPAG.LIS
DEACCESS_PENDING	[SYS.LIS]SYSQIOREQ.LIS
DEALLOC_DESCRIP	[SYS.LIS]IOSUBNPAG.LIS
DEALLOC_HQB	[MSCP.LIS]MSCP.LIS

Table C–2 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
DEANONPAGED	[DRIVER.LIS]DUTUSUBS.LIS
DEBUGGING_SANITY_CHECK	[SYS.LIS]IOSUBNPAG.LIS
DEFAULT_EB	[SYSLOA.LIS]SYS$SCS.LIS
DET_ACC_PATH	[MSCP.LIS]MSCP.LIS
DEVCHK	[SYS.LIS]IOSUBNPAG.LIS
DEV_NAME	[SYS.LIS]IOSUBNPAG.LIS
DGCOM	[DRIVER.LIS]SCSXPORT.LIS
DG_ALC_FAIL	[DRIVER.LIS]SCSXPORT.LIS
DIRECT	[SYS.LIS]SYSQIOREQ.LIS
DIRERR	[SYSLOA.LIS]SYS$SCS.LIS
DIR_CLEANUP	[SYSLOA.LIS]SYS$SCS.LIS
DIR_MOVE	[SYSLOA.LIS]SYS$SCS.LIS
DISPLAY_CTRLERR	[DRIVER.LIS]DUTUSUBS.LIS
DISPLAY_DRVERR	[DRIVER.LIS]DUTUSUBS.LIS
DISPLAY_IVCMD	[DRIVER.LIS]DUTUSUBS.LIS
DISPLAY_NAME	[SYS.LIS]IOSUBNPAG.LIS
DISPLAY_OFFLINE	[DRIVER.LIS]DUTUSUBS.LIS
DISPLAY_SUCC	[DRIVER.LIS]DUTUSUBS.LIS
DONE1	[SYSLOA.LIS]SYS$SCS.LIS
DONE2	[SYSLOA.LIS]SYS$SCS.LIS
DONE3	[SYSLOA.LIS]SYS$SCS.LIS
DONE	[SYSLOA.LIS]SYS$SCS.LIS
DONE_PDT	[SYSLOA.LIS]SYS$SCS.LIS
DONT_MV	[SYSLOA.LIS]MOUNTVER.LIS
DONT_MV_CK_SRVIO	[SYSLOA.LIS]MOUNTVER.LIS
DONT_MV_CK_SYS	[SYSLOA.LIS]MOUNTVER.LIS
DO_DISK	[MSCP.LIS]MSCP.LIS
DO_MAP	[DRIVER.LIS]DUDRIVER.LIS
DO_MV	[SYSLOA.LIS]MOUNTVER.LIS
DO_MV_VAL	[SYSLOA.LIS]MOUNTVER.LIS
DO_MV_VAL_ASSIST	[SYSLOA.LIS]MOUNTVER.LIS
DO_MV_WRITLCK	[SYSLOA.LIS]MOUNTVER.LIS
DO_ORIG_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DO_ORIG_UCB_IO	[DRIVER.LIS]DUTUSUBS.LIS
DO_PMS	[SYS.LIS]IOSUBNPAG.LIS

Table C–2 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
DO_RSB	[SYSLOA.LIS]SYS$SCS.LIS
DO_WRT	[MSCP.LIS]MSCP.LIS
DQUEUE_DG	[DRIVER.LIS]SCSXPORT.LIS
DQ_INCOMPLETE	[DRIVER.LIS]SCSXPORT.LIS
DRIVER_CODE	[SYSLOA.LIS]MOUNTVER.LIS
DSE_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
DU$CONNECT_ERR	[DRIVER.LIS]DUDRIVER.LIS
DU$CREDIT_STALL	[DRIVER.LIS]DUDRIVER.LIS
DU$DGDR	[DRIVER.LIS]DUDRIVER.LIS
DU$DSE_FDT	[DRIVER.LIS]DUDRIVER.LIS
DU$FUNCTION_EXIT	[DRIVER.LIS]DUDRIVER.LIS
DU$IDR	[DRIVER.LIS]DUDRIVER.LIS
DU$ILLIOFUNC	[DRIVER.LIS]DUDRIVER.LIS
DU$INVALID_STS	[DRIVER.LIS]DUDRIVER.LIS
DU$PATH_MOVE	[DRIVER.LIS]DUDRIVER.LIS
DU$RECORD_ONLINE	[DRIVER.LIS]DUDRIVER.LIS
DU$RECORD_UNIT_STATUS	[DRIVER.LIS]DUDRIVER.LIS
DU$RE_SYNCH	[DRIVER.LIS]DUDRIVER.LIS
DU$RE_SYNCH_PKT	[DRIVER.LIS]DUDRIVER.LIS
DU$SETPATH	[DRIVER.LIS]DUDRIVER.LIS
DU$SHAD_RWCHECK_FDT	[DRIVER.LIS]DUDRIVER.LIS
DU$TMR	[DRIVER.LIS]DUDRIVER.LIS
DU$TMR_BROKE	[DRIVER.LIS]DUDRIVER.LIS
DUPLICATE_SYSTEMID	[DRIVER.LIS]DUTUSUBS.LIS
DUPLICATE_UNIT_ATTN	[DRIVER.LIS]DUDRIVER.LIS
DUTU$BEGIN_CONN_WALK	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$BEGIN_MNTVER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$BUILD_CANIO_CDRP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CANCEL	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CANCEL_RDTWAIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CANCEL_RDT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CHECK_DDB_FOR_CDDB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CHECK_NOCANCEL	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CHECK_RWAITCNT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$CLEANUP_CANCEL	[DRIVER.LIS]DUTUSUBS.LIS

Table C–2 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
DUTU$CREATE_CDDB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DATA	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DEALLOC_ALL	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DEALLOC_RSPID_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DEVTYPE_TABLE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DISCONNECT_CANCEL	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DISPLAY	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DODAP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DO_CONN_WALKER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DRAIN_CDDB_CDRPQ	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DUMP_COMMAND	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$DUMP_ENDMESSAGE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$END_CANCEL	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$END_CONN_WALK	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$END_MNTVER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$END_SINGLE_STREAM	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$FAILOVER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$FILL_MSCP_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$FIND_DDB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$FIND_DDB_NOLOCK	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$GET_DEVNAM	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$GET_DEVTYPE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INIT_CONN_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INIT_MSCP_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INIT_MSCP_MSG_UNIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INSERT_RESTARTQ	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INTR_ACTION_N	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INTR_ACTION_XFER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INTR_SUB_N	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$INTR_SUB_XFER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$KILL_THIS_THREAD	[DRIVER.LIS]DUTUSUBS.LIS

Table C–3: VMS Routine and Module Cross Reference

Table C–3 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
Routine	**Module**
DUTU$LINK_SEC_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$LINK_UCB2CDDB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$LOCATE_UNIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$LOG_IVCMD	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$LOOKUP_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$LOOKUP_UCB_RESUME	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$MOUNTVER	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$MOVE_IODB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$MOVE_UNIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$MOVE_UNIT_NOSLUN	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$NEW_UNIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$OWN_STORAGE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$POLL_FOR_UNITS	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$POST_CDRP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$RECONN_LOOKUP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$RESET_MSCP_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$RESTART_NEXT_CDRP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$RESTORE_CREDIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$REVALIDATE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEND_DRIVER_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEND_DUPLICATE_UNIT	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEND_MSCP_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEND_WALKER_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SERVER_MV	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SETUP_CDP_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SETUP_DUAL_PATH	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEVER_CDDB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$SEVER_SEC_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$TEMPLATE_ORB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$TEMPLATE_UCB	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$TERMINATE_PENDING	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$TEST_CANCEL_CDRP	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$TEST_CANCEL_DONE	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$UNITINIT	[DRIVER.LIS]DUTUSUBS.LIS

Table C–3 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
DUTU$WAIT_ORIG_IO	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$WALK_NEXT_CONN	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$WALK_RESET_MSCP_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DUTU$WALK_SEND_MSG	[DRIVER.LIS]DUTUSUBS.LIS
DU_BEGIN_IVCMD	[DRIVER.LIS]DUDRIVER.LIS
DU_CONTROLLER_INIT	[DRIVER.LIS]DUDRIVER.LIS
DU_FUNCTABLE	[DRIVER.LIS]DUDRIVER.LIS
DU_RESTARTIO	[DRIVER.LIS]DUDRIVER.LIS
DU_STARTIO	[DRIVER.LIS]DUDRIVER.LIS
DU_UNSOLNT	[DRIVER.LIS]DUDRIVER.LIS
EMBEND1	[SYSLOA.LIS]SYS$SCS.LIS
EMBEND2	[SYSLOA.LIS]SYS$SCS.LIS
EMB_BUFFER	[SYSLOA.LIS]SYS$SCS.LIS
EMB_CURRENT	[SYSLOA.LIS]SYS$SCS.LIS
EMB_RED1	[SYSLOA.LIS]SYS$SCS.LIS
EMB_RED2	[SYSLOA.LIS]SYS$SCS.LIS
EMB_YELLOW1	[SYSLOA.LIS]SYS$SCS.LIS
EMB_YELLOW2	[SYSLOA.LIS]SYS$SCS.LIS
ENDDEALL	[SYSLOA.LIS]SYS$SCS.LIS
ENDPARENS	[SYS.LIS]IOSUBNPAG.LIS
ENDRED	[SYSLOA.LIS]SYS$SCS.LIS
ENDYELLOW	[SYSLOA.LIS]SYS$SCS.LIS
END_BROADCAST	[SYS.LIS]IOSUBNPAG.LIS
END_CONBRDCST	[SYS.LIS]IOSUBNPAG.LIS
END_IO	[SYSLOA.LIS]MOUNTVER.LIS
END_PACKACK	[DRIVER.LIS]DUDRIVER.LIS
END_PACKACK_BA	[DRIVER.LIS]DUDRIVER.LIS
ENTABLE_BEGIN	[SYSLOA.LIS]MOUNTVER.LIS
ENTABLE_END	[SYSLOA.LIS]MOUNTVER.LIS
EQUAL_PATH_CALL_COUNT	[SYSLOA.LIS]SYS$SCS.LIS
ERASE	[MSCP.LIS]MSCP.LIS
ERRORB	[SYS.LIS]SYSQIOREQ.LIS
ERROR	[SYS.LIS]SYSQIOREQ.LIS
ERROR_EXIT	[SYSLOA.LIS]MOUNTVER.LIS
ERROR_NO_UNIT	[MSCP.LIS]MSCP.LIS

Table C–3 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
ERROR_OUT	[MSCP.LIS]MSCP.LIS
ERR_OFFLINE	[MSCP.LIS]MSCP.LIS
ERR_ROUTINE	[SYSLOA.LIS]SYS$SCS.LIS
ERR_TBL	[MSCP.LIS]MSCP.LIS
ERR_WRITLCK	[MSCP.LIS]MSCP.LIS
EXDVNM	[SYS.LIS]IOSUBNPAG.LIS
EXE$ABORTIO	[SYS.LIS]SYSQIOREQ.LIS
EXE$ALTQUEPKT	[SYS.LIS]SYSQIOREQ.LIS
EXE$BLDPKTGSR	[SYS.LIS]SYSQIOREQ.LIS
EXE$BLDPKTGSW	[SYS.LIS]SYSQIOREQ.LIS
EXE$BLDPKTMPW	[SYS.LIS]SYSQIOREQ.LIS
EXE$BLDPKTSWPR	[SYS.LIS]SYSQIOREQ.LIS
EXE$BLDPKTSWPW	[SYS.LIS]SYSQIOREQ.LIS
EXE$BUILDPKTR	[SYS.LIS]SYSQIOREQ.LIS
EXE$BUILDPKTW	[SYS.LIS]SYSQIOREQ.LIS
EXE$CLUTRANIO	[SYSLOA.LIS]MOUNTVER.LIS
EXE$FINISHIOC	[SYS.LIS]SYSQIOREQ.LIS
EXE$FINISHIO	[SYS.LIS]SYSQIOREQ.LIS
EXE$INSERTIRP	[SYS.LIS]SYSQIOREQ.LIS
EXE$INSIOQC	[SYS.LIS]SYSQIOREQ.LIS
EXE$INSIOQ	[SYS.LIS]SYSQIOREQ.LIS
EXE$MATCH_NAME	[SYS.LIS]IOSUBNPAG.LIS
EXE$MNTVERSHDOL	[SYSLOA.LIS]MOUNTVER.LIS
EXE$MNTVERSIO	[SYSLOA.LIS]MOUNTVER.LIS
EXE$MNTVERSP1	[SYSLOA.LIS]MOUNTVER.LIS
EXE$MNTVERSP2	[SYSLOA.LIS]MOUNTVER.LIS
EXE$MNTVER_GEN_CRC	[SYSLOA.LIS]MOUNTVER.LIS
EXE$MOUNTVER	[SYSLOA.LIS]MOUNTVER.LIS
EXE$QIOACPPKT	[SYS.LIS]SYSQIOREQ.LIS
EXE$QIODRVPKT	[SYS.LIS]SYSQIOREQ.LIS
EXE$QIORETURN1	[SYS.LIS]SYSQIOREQ.LIS
EXE$QIORETURNL	[SYS.LIS]SYSQIOREQ.LIS
EXE$QIORETURN	[SYS.LIS]SYSQIOREQ.LIS
EXE$QXQPPKT	[SYS.LIS]SYSQIOREQ.LIS
EXE$UPDGNERNUM	[SYSLOA.LIS]MOUNTVER.LIS

Table C–3 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
EXITAVG11	[SYSLOA.LIS]SYS$SCS.LIS
EXIT	[SYSLOA.LIS]SYS$SCS.LIS
EXIT_ATTN_MSG	[DRIVER.LIS]DUDRIVER.LIS
EXIT_MOVE_IODB	[DRIVER.LIS]DUTUSUBS.LIS
EXTEND_HULB	[MSCP.LIS]MSCP.LIS

Table C–4: VMS Routine and Module Cross Reference

Routine	Module
FILL_BUFFER	[SYS.LIS]SYSACPFDT.LIS
FIND_BEST_MATCH	[SYSLOA.LIS]SYS$SCS.LIS
FIND_UQB	[MSCP.LIS]MSCP.LIS
FINISHED_WITH_MESSAGE	[DRIVER.LIS]DUDRIVER.LIS
FINISH_ORIGUCB	[DRIVER.LIS]DUTUSUBS.LIS
FKB_INIT	[MSCP.LIS]MSCP.LIS
FLUSH	[MSCP.LIS]MSCP.LIS
FORMAT_ABRTD	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_AVAIL	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_CDTERR	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_CTRLERR	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_DRVERR	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_HOST_BUFFER_ACCESS	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_HSTBF	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_IPARM	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_IVCMD	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_MFMTE	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_OFFLINE	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_PACKACK	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_SUCC	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_WRITE_LOCK	[DRIVER.LIS]DUDRIVER.LIS
FORMAT_WRTPR	[DRIVER.LIS]DUDRIVER.LIS
FPC$CHK_DCONID	[DRIVER.LIS]SCSXPORT.LIS
FPC$CHK_LCONID	[DRIVER.LIS]SCSXPORT.LIS
FPC$CHK_SCONID	[DRIVER.LIS]SCSXPORT.LIS

Table C–4 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
FPC_SUCCESS	[DRIVER.LIS]SCSXPORT.LIS
FREE_BUFFER	[SYSLOA.LIS]MOUNTVER.LIS
FULL	[MSCP.LIS]MSCP.LIS
FULL_NAME	[SYS.LIS]IOSUBNPAG.LIS
FUNCTION_EXIT	[DRIVER.LIS]DUDRIVER.LIS
FUNC_EXIT_REQCOM	[DRIVER.LIS]DUDRIVER.LIS
GENCRC_PACKACK	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_POSITION	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_READ	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_REWIND	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_UNLOAD	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_WRITEOF	[SYSLOA.LIS]MOUNTVER.LIS
GENCRC_WRITE	[SYSLOA.LIS]MOUNTVER.LIS
GETNUMBER	[SYS.LIS]IOSUBNPAG.LIS
GET_BUFFER	[SYSLOA.LIS]MOUNTVER.LIS
GET_CHECKSUM_BUF	[SYSLOA.LIS]MOUNTVER.LIS
GET_COMMAND_STATUS	[MSCP.LIS]MSCP.LIS
GET_FIRST_SB	[SYS.LIS]IOSUBNPAG.LIS
GET_FIRST_SB_LNK	[SYS.LIS]IOSUBNPAG.LIS
GET_MSG_ID	[SYSLOA.LIS]MOUNTVER.LIS
GET_NEXT_DDB	[SYS.LIS]IOSUBNPAG.LIS
GET_NEXT_SB	[SYS.LIS]IOSUBNPAG.LIS
GET_PARENT_SB	[SYS.LIS]IOSUBNPAG.LIS
GET_UNIT_STATUS	[MSCP.LIS]MSCP.LIS
GET_VCB	[SYSLOA.LIS]MOUNTVER.LIS
GTPKT	[SYS.LIS]SYSQIOREQ.LIS
ILLIO	[SYS.LIS]SYSQIOREQ.LIS
IMMEDIATE	[MSCP.LIS]MSCP.LIS
INCR_SANITY_LOOP	[SYS.LIS]IOSUBNPAG.LIS
INI$IOC_RETURN	[SYS.LIS]IOSUBNPAG.LIS
INITIAL	[SYS.LIS]IOSUBNPAG.LIS
INIT_CDL	[SYSLOA.LIS]SYS$SCS.LIS
INIT_DONE	[MSCP.LIS]MSCP.LIS
INIT_IRP	[SYSLOA.LIS]MOUNTVER.LIS
INIT_IRP_READ	[SYSLOA.LIS]MOUNTVER.LIS

Table C–4 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
INIT_IRP_REPOS	[SYSLOA.LIS]MOUNTVER.LIS
INIT_IRP_REWIND	[SYSLOA.LIS]MOUNTVER.LIS
INIT_LOAD_SHARE	[SYSLOA.LIS]SYS$SCS.LIS
INIT_RDT	[SYSLOA.LIS]SYS$SCS.LIS
INIT_TIMEOUT	[DRIVER.LIS]DUDRIVER.LIS
INTR_ACTION_COMMON	[DRIVER.LIS]DUTUSUBS.LIS
INTR_SUB_ACTION_COMMON	[DRIVER.LIS]DUTUSUBS.LIS
INTR_SUB_COMMON	[DRIVER.LIS]DUTUSUBS.LIS
INVALID_STS	[DRIVER.LIS]DUDRIVER.LIS
INVAL	[MSCP.LIS]MSCP.LIS
INV_ATTN_MSG	[DRIVER.LIS]DUDRIVER.LIS
INV_FLAGS	[MSCP.LIS]MSCP.LIS
IOC$ALODATAP	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOMAPUDA	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOUBAMAPN	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOUBAMAPSP	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOUBAMAP	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOUBMAPRMN	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALOUBMAPRM	[SYS.LIS]IOSUBNPAG.LIS
IOC$ALTREQCOM	[SYS.LIS]IOSUBNPAG.LIS
IOC$APPLYECC	[SYS.LIS]IOSUBRAMS.LIS
IOC$BROADCAST	[SYS.LIS]IOSUBNPAG.LIS
IOC$CANCELIO	[SYS.LIS]IOSUBNPAG.LIS
IOC$CHECK_HWM	[SYS.LIS]SYSACPFDT.LIS
IOC$CONBRDCST	[SYS.LIS]IOSUBNPAG.LIS
IOC$CTRLINIT	[SYS.LIS]IOSUBNPAG.LIS

Table C–5: VMS Routine and Module Cross Reference

Routine	Module
IOC$CVTLOGPHY	[SYS.LIS]IOSUBRAMS.LIS
IOC$CVTLOGPHYU	[SYS.LIS]IOSUBRAMS.LIS
IOC$CVT_DEVNAM	[SYS.LIS]IOSUBNPAG.LIS
IOC$DALOCUBAMAP	[SYS.LIS]IOSUBNPAG.LIS
IOC$DCLSYSEVT	[SYS.LIS]IOSUBNPAG.LIS

Table C–5 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
IOC$DIAGBUFILL	[SYS.LIS]IOSUBNPAG.LIS
IOC$INITIATE	[SYS.LIS]IOSUBNPAG.LIS
IOC$LAST_CHAN	[SYS.LIS]IOSUBNPAG.LIS
IOC$LAST_CHAN_AMBX	[SYS.LIS]IOSUBNPAG.LIS
IOC$LOG_EVENT	[SYS.LIS]IOSUBNPAG.LIS
IOC$MAPVBLK	[SYS.LIS]IOSUBRAMS.LIS
IOC$MNTVER	[SYS.LIS]IOSUBNPAG.LIS
IOC$PARSDEVNAM	[SYS.LIS]IOSUBNPAG.LIS
IOC$POST_IRP	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELCHAN	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELDATAPUDA	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELDATAP	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELMAPREG	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELMAPUDA	[SYS.LIS]IOSUBNPAG.LIS
IOC$RELSCHAN	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQCOM	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQDATAPNW	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQDATAPUDA	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQDATAP	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQMAPREG	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQMAPUDA	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQPCHANH	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQPCHANL	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQSCHANH	[SYS.LIS]IOSUBNPAG.LIS
IOC$REQSCHANL	[SYS.LIS]IOSUBNPAG.LIS
IOC$SCAN_IODB	[SYS.LIS]IOSUBNPAG.LIS
IOC$SCAN_IODB_2P	[SYS.LIS]IOSUBNPAG.LIS
IOC$SCAN_IODB_USRCTX	[SYS.LIS]IOSUBNPAG.LIS
IOC$SEARCHCONT	[SYS.LIS]IOSUBNPAG.LIS
IOC$SEARCHINT	[SYS.LIS]IOSUBNPAG.LIS
IOC$SENSEDISK	[SYS.LIS]IOSUBRAMS.LIS
IOC$TESTUNIT	[SYS.LIS]IOSUBNPAG.LIS
IOC$THREADCRB	[SYS.LIS]IOSUBNPAG.LIS
IOC$UNITINIT	[SYS.LIS]IOSUBNPAG.LIS
IOC$UPDATRANSP	[SYS.LIS]IOSUBRAMS.LIS

Table C–5 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
IOC$WFIKPCH	[SYS.LIS]IOSUBNPAG.LIS
IOC$WFIRLCH	[SYS.LIS]IOSUBNPAG.LIS
IO_STALLED	[DRIVER.LIS]DUDRIVER.LIS
IO_STALLED_BA	[DRIVER.LIS]DUDRIVER.LIS
IRP_ALLOC_ERR	[MSCP.LIS]MSCP.LIS
IVCHAN	[SYS.LIS]SYSQIOREQ.LIS
IVCMD_ALIGN	[DRIVER.LIS]DUTUSUBS.LIS
IVCMD_ENDMSG	[DRIVER.LIS]DUTUSUBS.LIS
IVCMD_MSGLEN	[DRIVER.LIS]DUTUSUBS.LIS
IVCMD_ORGMSG	[DRIVER.LIS]DUTUSUBS.LIS
IVCMD_WORK	[DRIVER.LIS]DUTUSUBS.LIS
KFMSA_EB	[SYSLOA.LIS]SYS$SCS.LIS
LEAST_LOADED	[SYSLOA.LIS]SYS$SCS.LIS
LINK_2P_UCB	[DRIVER.LIS]DUTUSUBS.LIS
LINK_NEW_UCB	[DRIVER.LIS]DUTUSUBS.LIS
LISTENERR	[SYSLOA.LIS]SYS$SCS.LIS
LISTN	[MSCP.LIS]MSCP.LIS
LIS_ERR	[MSCP.LIS]MSCP.LIS
LM_DONE	[MSCP.LIS]MSCP.LIS
LM_EXIT	[MSCP.LIS]MSCP.LIS
LM_INIT	[MSCP.LIS]MSCP.LIS
LM_INIT_CAPACITY	[MSCP.LIS]MSCP.LIS
LM_LOOP	[MSCP.LIS]MSCP.LIS
LM_PASS	[MSCP.LIS]MSCP.LIS
LOAD_DEFAULT	[MSCP.LIS]MSCP.LIS
LOAD_MONITOR	[MSCP.LIS]MSCP.LIS
LOAD_SHARE_BEG_TIME	[SYSLOA.LIS]SYS$SCS.LIS
LOAD_SHARE_BUGCHECK	[SYSLOA.LIS]SYS$SCS.LIS
LOAD_SHARE_BUGCHECK_COUNT	[SYSLOA.LIS]SYS$SCS.LIS
LOAD_SHARE_END_TIME	[SYSLOA.LIS]SYS$SCS.LIS
LOCAL_NAME	[SYS.LIS]IOSUBNPAG.LIS
LOCATE_WBUF	[SYSLOA.LIS]MOUNTVER.LIS
LOCKERR	[SYS.LIS]SYSACPFDT.LIS
LOG_ATTENTION_MESSAGE	[DRIVER.LIS]DUDRIVER.LIS
LOG_CMD_PKT	[MSCP.LIS]MSCP.LIS

Table C–5 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
LOG_END_PKT	[MSCP.LIS]MSCP.LIS
LOG_FINAL_STATUS	[DRIVER.LIS]DUDRIVER.LIS
LOOKUP_LOCAL_UCB	[DRIVER.LIS]DUTUSUBS.LIS
LOOP1	[SYSLOA.LIS]SYS$SCS.LIS
LOOP2	[SYSLOA.LIS]SYS$SCS.LIS
LOOP3	[SYSLOA.LIS]SYS$SCS.LIS
LS_DOMAIN_PORTS	[SYSLOA.LIS]SYS$SCS.LIS
MAKE_CONNECTION	[DRIVER.LIS]DUDRIVER.LIS
MAP_COMMON	[DRIVER.LIS]SCSXPORT.LIS
MAP_USER_BUFFER	[SYSLOA.LIS]MOUNTVER.LIS
MNTVERPNDCHK	[SYS.LIS]IOSUBNPAG.LIS
MOST_LOADED	[SYSLOA.LIS]SYS$SCS.LIS
MSCP$ALLOCATE	[MSCP.LIS]MSCP.LIS
MSCP$DEALLOCATE	[MSCP.LIS]MSCP.LIS
MSCP$TMR	[MSCP.LIS]MSCP.LIS
MSCP_BEGIN	[MSCP.LIS]MSCP.LIS
MSCP_BUF	[DRIVER.LIS]DUTUSUBS.LIS
MSCP_END	[MSCP.LIS]MSCP.LIS
MSCP_START	[MSCP.LIS]MSCP.LIS
MSG_BUF_FAILURE	[DRIVER.LIS]DUDRIVER.LIS
MSG_INPUT	[SYSLOA.LIS]SYS$SCS.LIS
MSG_IN	[MSCP.LIS]MSCP.LIS
MV_INITIATE	[SYSLOA.LIS]MOUNTVER.LIS
MV_MSCP	[SYSLOA.LIS]MOUNTVER.LIS
MV_RSB	[SYSLOA.LIS]MOUNTVER.LIS
MV_SET_OFFLINE	[MSCP.LIS]MSCP.LIS

Table C–6: VMS Routine and Module Cross Reference

Routine	Module
NALLOC	[SYS.LIS]SYSQIOREQ.LIS
NATIVE_SNDDATWM	[DRIVER.LIS]SCSXPORT.LIS
NEWMODE	[SYS.LIS]SYSACPFDT.LIS
NEW_DEVICE	[MSCP.LIS]MSCP.LIS
NEXTUCB	[SYS.LIS]IOSUBNPAG.LIS

Table C–6 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
NEXT_REVAL_UCB	[DRIVER.LIS]DUTUSUBS.LIS
NOACP	[SYS.LIS]SYSACPFDT.LIS
NOCNT	[SYS.LIS]SYSQIOREQ.LIS
NODCNT	[SYS.LIS]SYSQIOREQ.LIS
NOIOSB	[SYS.LIS]SYSQIOREQ.LIS
NOMEM	[MSCP.LIS]MSCP.LIS
NOMEM_ERR	[MSCP.LIS]MSCP.LIS
NONSEQB	[MSCP.LIS]MSCP.LIS
NONSEQ	[MSCP.LIS]MSCP.LIS
NOP_AVAIL	[DRIVER.LIS]DUDRIVER.LIS
NOP_CDTERR	[DRIVER.LIS]DUDRIVER.LIS
NOP_CTRLERR	[DRIVER.LIS]DUDRIVER.LIS
NOP_DRVERR	[DRIVER.LIS]DUDRIVER.LIS
NOP_IVCMD	[DRIVER.LIS]DUDRIVER.LIS
NOP_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
NOP_OFFLINE	[DRIVER.LIS]DUDRIVER.LIS
NOP_SSSC	[DRIVER.LIS]DUDRIVER.LIS
NOP_SUCC	[DRIVER.LIS]DUDRIVER.LIS
NORMAL	[MSCP.LIS]MSCP.LIS
NORMAL_TRANSFEREND	[DRIVER.LIS]DUDRIVER.LIS
NORMAL_ZONE	[SYSLOA.LIS]SYS$SCS.LIS
NORM_EXIT	[SYSLOA.LIS]MOUNTVER.LIS
NOSECT	[SYS.LIS]SYSQIOREQ.LIS
NOSERV	[MSCP.LIS]MSCP.LIS
NOTSHR	[MSCP.LIS]MSCP.LIS
NOT_FILE_DEVB	[SYS.LIS]SYSQIOREQ.LIS
NOT_FILE_DEV	[SYS.LIS]SYSQIOREQ.LIS
NOT_FIRST	[SYS.LIS]IOSUBNPAG.LIS
NOT_NEXT_UNIT	[SYS.LIS]IOSUBNPAG.LIS
NOT_RCT_ACCESS	[DRIVER.LIS]DUDRIVER.LIS
NOT_READY	[DRIVER.LIS]SCSXPORT.LIS
NO_CDT	[SYSLOA.LIS]SYS$SCS.LIS
NO_MEM	[MSCP.LIS]MSCP.LIS
NO_MSCP_LISTN	[SYSLOA.LIS]SYS$SCS.LIS
NO_ORIGUCB	[DRIVER.LIS]DUTUSUBS.LIS

Table C–6 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
NO_ROOM	[MSCP.LIS]MSCP.LIS
NO_SECONDARY	[SYS.LIS]IOSUBNPAG.LIS
NSPOOL	[SYS.LIS]SYSQIOREQ.LIS
NUM_DISCONNECT	[SYSLOA.LIS]SYS$SCS.LIS
NXTIRP	[SYS.LIS]IOSUBNPAG.LIS
OFFLINE	[SYS.LIS]SYSQIOREQ.LIS
OH_NO	[DRIVER.LIS]DUTUSUBS.LIS
OK	[SYS.LIS]SYSQIOREQ.LIS
OK_TEST_LAST_REG	[SYS.LIS]IOSUBNPAG.LIS
ONLINE	[MSCP.LIS]MSCP.LIS
PACKACK_576	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_ABORT	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_AVLBL	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_BADRCT	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_CANCEL	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_COUNT_MEMBERS	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_DO_ONLINE	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_DUPUN	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_FAILOVER	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_FAILOVER_MSG	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_FAIL	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_GTUNT_SUCC	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_IVCMD	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_MFMTE	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_MOVE_SERVER	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_OFFLINE	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_SUCC	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_TEST_SSM	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_UNAVAILABLE	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_UNKNO_INOPR	[DRIVER.LIS]DUDRIVER.LIS
PACKACK_VOLUME	[SYSLOA.LIS]MOUNTVER.LIS
PACKACK_WRONG_D0D1	[DRIVER.LIS]DUDRIVER.LIS
PACKET_ERROR	[MSCP.LIS]MSCP.LIS
PATCH_DEBUGGING_SANITY_CHECK	[SYS.LIS]IOSUBNPAG.LIS
PAUSE	[SYSLOA.LIS]MOUNTVER.LIS

Table C–6 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
PCHIST	[MSCP.LIS]MSCP.LIS
PDTDONE	[SYSLOA.LIS]SYS$SCS.LIS
PDT_JMP	[SYSLOA.LIS]SYS$SCS.LIS
PERFORM_SHADOW	[SYSLOA.LIS]MOUNTVER.LIS
PERFORM_VALIDATE	[SYSLOA.LIS]MOUNTVER.LIS
PHYIO_VOLINV	[DRIVER.LIS]DUDRIVER.LIS
PHYS_BAD_BCNT	[DRIVER.LIS]DUDRIVER.LIS
PHYS_BAD_LBN	[DRIVER.LIS]DUDRIVER.LIS
PHYS_IO_NORCT	[DRIVER.LIS]DUDRIVER.LIS
PHYS_IO_RCT	[DRIVER.LIS]DUDRIVER.LIS
PKAK_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
PMSEND	[SYS.LIS]IOSUBNPAG.LIS
POLLER_INIT	[SYSLOA.LIS]SYS$SCS.LIS
POLL_CONN_BROKE	[DRIVER.LIS]DUTUSUBS.LIS
POLL_LOOP	[DRIVER.LIS]DUTUSUBS.LIS
PORT_FAIR_SHARE	[SYSLOA.LIS]SYS$SCS.LIS
PRIOSB	[SYS.LIS]SYSQIOREQ.LIS
PRIVERR	[SYS.LIS]SYSQIOREQ.LIS
PRP_STCON_MSG	[DRIVER.LIS]DUDRIVER.LIS
PUTASCIC	[SYS.LIS]IOSUBNPAG.LIS
PUTCHAR	[SYS.LIS]IOSUBNPAG.LIS
PUTDOLLAR	[SYS.LIS]IOSUBNPAG.LIS
PUTNUM	[SYS.LIS]IOSUBNPAG.LIS
PUTSPACE	[SYS.LIS]IOSUBNPAG.LIS

Table C–7: VMS Routine and Module Cross Reference

Routine	Module
QIORETURN	[SYS.LIS]SYSQIOREQ.LIS
QIO	[SYS.LIS]SYSQIOREQ.LIS
QUEUE_DG	[DRIVER.LIS]SCSXPORT.LIS
QUEUE_IRP	[DRIVER.LIS]DUDRIVER.LIS
Q_FIPL	[SYS.LIS]SYSQIOREQ.LIS
Q_INCOMPLETE	[DRIVER.LIS]SCSXPORT.LIS
Q_SUCCESS	[DRIVER.LIS]SCSXPORT.LIS

Table C–7 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
RATE_DONE	[SYSLOA.LIS]SYS$SCS.LIS
RDDESCR	[SYS.LIS]SYSACPFDT.LIS
RD_SEQ_ERR	[DRIVER.LIS]SCSXPORT.LIS
READ	[MSCP.LIS]MSCP.LIS
READ_ACCESS	[SYS.LIS]SYSQIOREQ.LIS
READ_LOOP	[MSCP.LIS]MSCP.LIS
REALLOC_CD_MAPREGS	[SYS.LIS]IOSUBNPAG.LIS
RECONN_COMMON2	[DRIVER.LIS]DUDRIVER.LIS
RECONN_COMMON	[DRIVER.LIS]DUDRIVER.LIS
RECONN_EXIT	[DRIVER.LIS]DUTUSUBS.LIS
RECORD_ONLINE	[DRIVER.LIS]DUDRIVER.LIS
RECORD_STCON	[DRIVER.LIS]DUDRIVER.LIS
RECORD_UNIT_STATUS	[DRIVER.LIS]DUDRIVER.LIS
RELDATAP_COMMON	[SYS.LIS]IOSUBNPAG.LIS
RELEASE	[SYS.LIS]IOSUBNPAG.LIS
REPLACE	[MSCP.LIS]MSCP.LIS
RESET_TIMER	[SYSLOA.LIS]SYS$SCS.LIS
RESTARTIO	[DRIVER.LIS]DUDRIVER.LIS
RESTART_FIRST_CDRP	[DRIVER.LIS]DUDRIVER.LIS
RESTART_POLL	[DRIVER.LIS]DUTUSUBS.LIS
RESTOR	[MSCP.LIS]MSCP.LIS
REVAL_HBS_MEMBER	[DRIVER.LIS]DUTUSUBS.LIS
REVAL_STARTED	[DRIVER.LIS]DUTUSUBS.LIS
SANITY_ALL_DONE	[SYS.LIS]IOSUBNPAG.LIS
SANITY_LOOP	[SYS.LIS]IOSUBNPAG.LIS
SCAN_ALL_DEVICES	[BOOTS.LIS]STACONFIG_MSCP.LIS
SCS$ACCEPT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$ALLOC_CDT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$ALLOC_RSPID	[SYSLOA.LIS]SYS$SCS.LIS
SCS$CANCEL_MBX	[SYSLOA.LIS]SYS$SCS.LIS
SCS$CHECK_POOL	[DRIVER.LIS]SCSXPORT.LIS
SCS$CONFIG_PTH	[SYSLOA.LIS]SYS$SCS.LIS
SCS$CONFIG_SYS	[SYSLOA.LIS]SYS$SCS.LIS
SCS$CONNECT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$DEALL_CDT1	[SYSLOA.LIS]SYS$SCS.LIS

Table C–7 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
SCS$DEALL_CDT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$DEALL_RSPID	[SYSLOA.LIS]SYS$SCS.LIS
SCS$DIRECTORY	[SYSLOA.LIS]SYS$SCS.LIS
SCS$DIR_LOOKUP	[SYSLOA.LIS]SYS$SCS.LIS
SCS$DISCONNECT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$ENTER	[SYSLOA.LIS]SYS$SCS.LIS
SCS$FIND_RDTE	[SYSLOA.LIS]SYS$SCS.LIS
SCS$FPC_ALLOCDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_ALLOCMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_DEALLOCDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_DEALLOMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_DEALRGMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_MAPBYPASS	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_MAPIRPBYP	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_MAPIRP	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_MAP	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_QUEUEDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_QUEUEMDGS	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_RCHMSGBUF	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_RCLMSGBUF	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_REQDATA	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SCAN_MAP_WAIT	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SENDDATAWMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SENDDATA	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SENDDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SENDMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SENDRGDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_SNDCNTMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$FPC_UNMAP	[DRIVER.LIS]SCSXPORT.LIS
SCS$FREE_MSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$GL_LOAD_SHARE_COUNTERS	[SYSLOA.LIS]SYS$SCS.LIS
SCS$INITIAL	[DRIVER.LIS]SCSXPORT.LIS
SCS$LISTEN	[SYSLOA.LIS]SYS$SCS.LIS
SCS$LKP_MSGWAIT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$LKP_RDTCDRP	[SYSLOA.LIS]SYS$SCS.LIS

Table C–7 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
SCS$LKP_RDTWAIT	[SYSLOA.LIS]SYS$SCS.LIS
SCS$LOCLOOKUP	[SYSLOA.LIS]SYS$SCS.LIS
SCS$L_EB_TBL	[SYSLOA.LIS]SYS$SCS.LIS
SCS$NEW_PB	[SYSLOA.LIS]SYS$SCS.LIS
SCS$NEW_SB	[SYSLOA.LIS]SYS$SCS.LIS
SCS$POLL_MBX	[SYSLOA.LIS]SYS$SCS.LIS
SCS$POLL_MODE	[SYSLOA.LIS]SYS$SCS.LIS
SCS$POLL_PROC	[SYSLOA.LIS]SYS$SCS.LIS
SCS$RECYL_RSPID	[SYSLOA.LIS]SYS$SCS.LIS
SCS$REC_CNFREC	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_CNFWMREC	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_DATREC	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_DGREC	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_MSGREC	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_SNDDG	[DRIVER.LIS]SCSXPORT.LIS
SCS$REC_SNDMSG	[DRIVER.LIS]SCSXPORT.LIS
SCS$REMOVE	[SYSLOA.LIS]SYS$SCS.LIS
SCS$RESUMEWAITR	[SYSLOA.LIS]SYS$SCS.LIS
SCS$SET_LOAD_RATING	[SYSLOA.LIS]SYS$SCS.LIS
SCS$SHUTDOWN	[SYSLOA.LIS]SYS$SCS.LIS
SCS$UNSTALLUCB	[SYSLOA.LIS]SYS$SCS.LIS
SCS$UPDATE_PORT_LOAD_VECTOR	[SYSLOA.LIS]SYS$SCS.LIS
SCS_ALONONPAGED	[SYSLOA.LIS]SYS$SCS.LIS
SCS_CALC_BYTES_XFER	[SYSLOA.LIS]SYS$SCS.LIS
SCS_CHECK_QUEUES	[SYSLOA.LIS]SYS$SCS.LIS

Table C–8: VMS Routine and Module Cross Reference

Routine	Module
SCS_CHK_CDT_REUSE_QUEUE	[SYSLOA.LIS]SYS$SCS.LIS
SCS_CONTROL_MSG	[DRIVER.LIS]SCSXPORT.LIS
SCS_CTRS_UPDATE	[SYSLOA.LIS]SYS$SCS.LIS
SCS_DEALNONPAGD	[SYSLOA.LIS]SYS$SCS.LIS
SCS_END	[SYSLOA.LIS]SYS$SCS.LIS
SCS_GET_LOAD_VECTOR	[SYSLOA.LIS]SYS$SCS.LIS

Table C–8 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
SCS_INIT	[SYSLOA.LIS]SYS$SCS.LIS
SCS_LOAD_SHARE_TIMER	[SYSLOA.LIS]SYS$SCS.LIS
SCS_L_SHARE_TQE	[SYSLOA.LIS]SYS$SCS.LIS
SCS_PROCESS_CDTS	[SYSLOA.LIS]SYS$SCS.LIS
SCS_REDISTRIBUTE_CDTS	[SYSLOA.LIS]SYS$SCS.LIS
SCS_REGISTER_LOCAL_NAME	[SYSLOA.LIS]SYS$SCS.LIS
SCS_SELECT_PATH	[SYSLOA.LIS]SYS$SCS.LIS
SCS_UPDATE_REG_CTRS	[SYSLOA.LIS]SYS$SCS.LIS
SCS_VECTOR_FLAG	[SYSLOA.LIS]SYS$SCS.LIS
SC_SEQ_ERR	[DRIVER.LIS]SCSXPORT.LIS
SDONE1	[SYSLOA.LIS]SYS$SCS.LIS
SDONE	[SYSLOA.LIS]SYS$SCS.LIS
SECONDARY_NAME	[SYS.LIS]IOSUBNPAG.LIS
SECTION	[SYS.LIS]SYSQIOREQ.LIS
SEND_ABORTS	[DRIVER.LIS]DUTUSUBS.LIS
SEND_AVAIL	[DRIVER.LIS]DUDRIVER.LIS
SEND_END	[MSCP.LIS]MSCP.LIS
SEND_MESSAGE	[SYSLOA.LIS]MOUNTVER.LIS
SEND_PKT	[MSCP.LIS]MSCP.LIS
SEQUENTIAL	[MSCP.LIS]MSCP.LIS
SEQ_ALT	[MSCP.LIS]MSCP.LIS
SETUP_ERASE	[SYS.LIS]SYSACPFDT.LIS
SET_CONTROLLER_CHAR	[MSCP.LIS]MSCP.LIS
SET_RATING	[SYSLOA.LIS]SYS$SCS.LIS
SET_UNIT_CHR	[MSCP.LIS]MSCP.LIS
SHARE_TIMER_COUNT	[SYSLOA.LIS]SYS$SCS.LIS
SHDIO_REJECT	[DRIVER.LIS]DUDRIVER.LIS
SINGLE_STREAM	[DRIVER.LIS]DUDRIVER.LIS
SLOOP	[SYSLOA.LIS]SYS$SCS.LIS
SORTED	[SYS.LIS]IOSUBNPAG.LIS
SPOOL	[SYS.LIS]SYSQIOREQ.LIS
STALE_CDT	[DRIVER.LIS]SCSXPORT.LIS
START	[MSCP.LIS]MSCP.LIS
START_AVAILABLE	[DRIVER.LIS]DUDRIVER.LIS
START_DISPLAY	[DRIVER.LIS]DUTUSUBS.LIS

Table C–8 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
START_DSE	[DRIVER.LIS]DUDRIVER.LIS
START_FORMAT	[DRIVER.LIS]DUDRIVER.LIS
START_LOCAL_DEVICE	[DRIVER.LIS]DUDRIVER.LIS
START_MOUNT_VER	[DRIVER.LIS]DUDRIVER.LIS
START_NOP	[DRIVER.LIS]DUDRIVER.LIS
START_NOSUCH	[DRIVER.LIS]DUDRIVER.LIS
START_PACKACK	[DRIVER.LIS]DUDRIVER.LIS
START_POLL	[SYSLOA.LIS]SYS$SCS.LIS
START_READPBLK	[DRIVER.LIS]DUDRIVER.LIS
START_READWRITE	[DRIVER.LIS]DUDRIVER.LIS
START_SETPATH	[DRIVER.LIS]DUDRIVER.LIS
START_UNLOAD	[DRIVER.LIS]DUDRIVER.LIS
START_WCHK_DSE	[DRIVER.LIS]DUDRIVER.LIS
START_WITH_MODIFIERS	[DRIVER.LIS]DUDRIVER.LIS
START_WRITECHECK	[DRIVER.LIS]DUDRIVER.LIS
START_WRITEPBLK	[DRIVER.LIS]DUDRIVER.LIS
STATE_ERR	[DRIVER.LIS]SCSXPORT.LIS
STATE_ERR_R3	[DRIVER.LIS]SCSXPORT.LIS
STMV_BA	[DRIVER.LIS]DUDRIVER.LIS
SUCCES	[SYS.LIS]SYSQIOREQ.LIS
SVCDRP	[DRIVER.LIS]DUTUSUBS.LIS
SVERLIP	[DRIVER.LIS]DUTUSUBS.LIS
SVPDT	[DRIVER.LIS]DUTUSUBS.LIS
SVR7	[DRIVER.LIS]DUTUSUBS.LIS
SVRET	[DRIVER.LIS]DUTUSUBS.LIS
SVSTATUS	[DRIVER.LIS]DUTUSUBS.LIS
SVUCB	[DRIVER.LIS]DUTUSUBS.LIS
TEST_QUORUM	[SYSLOA.LIS]MOUNTVER.LIS
THIS_ISNT_SORTED	[SYS.LIS]IOSUBNPAG.LIS
THREAD_HAS_RWAITCNT	[SYSLOA.LIS]SYS$SCS.LIS
TIME_DELAY	[SYSLOA.LIS]MOUNTVER.LIS
TRANSFER_CDTERR	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_DATA_ERROR	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_HOST_BUFFER_ERROR	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_INVALID_COMMAND	[DRIVER.LIS]DUDRIVER.LIS

Table C–8 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
TRANSFER_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_MEDOFL	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_MSCP_ERROR	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_REPLACE	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_RTN_BCNT	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_SHIFT	[DRIVER.LIS]DUDRIVER.LIS
TRANSFER_SIZE	[SYSLOA.LIS]SYS$SCS.LIS
UCB_SCAN_LOOP	[SYS.LIS]IOSUBNPAG.LIS
UNBLOCK	[MSCP.LIS]MSCP.LIS
UNDO_ONLINE	[DRIVER.LIS]DUDRIVER.LIS
UNDO_ONLINE_BA	[DRIVER.LIS]DUDRIVER.LIS
UNEQUAL_PATH_CALL_COUNT	[SYSLOA.LIS]SYS$SCS.LIS
UNHOOK_CDRP	[MSCP.LIS]MSCP.LIS
UNIT_AVAILABLE_ATTN	[DRIVER.LIS]DUDRIVER.LIS
UNSORTED	[SYS.LIS]IOSUBNPAG.LIS
UNSORTED_START	[SYS.LIS]IOSUBNPAG.LIS
UPBYTCNT	[SYS.LIS]SYSACPFDT.LIS
UPBYTCT1	[SYS.LIS]SYSACPFDT.LIS
UPDATE_AVAIL_MAP	[SYSLOA.LIS]SYS$SCS.LIS
UPDATE_SCB	[SYSLOA.LIS]MOUNTVER.LIS
UPTRANSCNT	[SYS.LIS]SYSACPFDT.LIS

Table C–9: VMS Routine and Module Cross Reference

Routine	Module
VALIDATE_BAD_VOLNAM	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_ERROR	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_EXIT	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_HOME	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_SCB	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_SUCCESS	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_TAPE	[SYSLOA.LIS]MOUNTVER.LIS
VALIDATE_VOLUME	[SYSLOA.LIS]MOUNTVER.LIS
VALID_PACKACK	[DRIVER.LIS]DUDRIVER.LIS
VCHAN	[SYS.LIS]SYSQIOREQ.LIS

Table C–9 (Cont.): VMS Routine and Module Cross Reference

Routine	Module
VC_ERR	[MSCP.LIS]MSCP.LIS
VC_ERR_ABORT	[MSCP.LIS]MSCP.LIS
VC_ERR_COMMON	[MSCP.LIS]MSCP.LIS
VC_PATH_MOVE	[MSCP.LIS]MSCP.LIS
VOLNOTVAL	[DRIVER.LIS]DUDRIVER.LIS
VOL_INVALID	[DRIVER.LIS]DUDRIVER.LIS
WAIT_BD	[DRIVER.LIS]SCSXPORT.LIS
WAIT_FOR_INT	[MSCP.LIS]MSCP.LIS
WAIT_FOR_QUORUM	[SYSLOA.LIS]MOUNTVER.LIS
WALKERR	[DRIVER.LIS]DUTUSUBS.LIS
WALK_NEXT_COMMON	[DRIVER.LIS]DUTUSUBS.LIS
WALK_TEST_CONN	[DRIVER.LIS]DUTUSUBS.LIS
WLOOP	[SYSLOA.LIS]SYS$SCS.LIS
WORSE_PATH_CONNECTIONS	[SYSLOA.LIS]SYS$SCS.LIS
WRDESCR	[SYS.LIS]SYSACPFDT.LIS
WRITE	[MSCP.LIS]MSCP.LIS
WRITE_ACCESS	[SYS.LIS]SYSQIOREQ.LIS
WRITE_LOOP	[MSCP.LIS]MSCP.LIS
WRITLCK_HNDLR	[SYSLOA.LIS]MOUNTVER.LIS
XFER_ERR	[MSCP.LIS]MSCP.LIS
XFER_IVCMD_END	[DRIVER.LIS]DUDRIVER.LIS
XFER_NORMALEND	[DRIVER.LIS]DUDRIVER.LIS
XFER_REPLACE	[DRIVER.LIS]DUDRIVER.LIS
XFER_RTN_BCNT	[DRIVER.LIS]DUDRIVER.LIS
XFER_SETUP_ERROR	[DRIVER.LIS]DUDRIVER.LIS
XQP$BLOCK_ROUTINE	[SYS.LIS]SYSACPFDT.LIS
XQP$FCBSTALE	[SYS.LIS]SYSACPFDT.LIS
XQP$REL_QUOTA	[SYS.LIS]SYSACPFDT.LIS
XQP$UNLOCK_CACHE	[SYS.LIS]SYSACPFDT.LIS
XQP	[SYS.LIS]SYSQIOREQ.LIS
XQUOTA	[SYS.LIS]SYSACPFDT.LIS
X_PHYS_IO_RCT	[DRIVER.LIS]DUDRIVER.LIS
YELLOWQ	[SYSLOA.LIS]SYS$SCS.LIS
Y_ZONE	[SYSLOA.LIS]SYS$SCS.LIS
ZERO_TOL	[SYSLOA.LIS]SYS$SCS.LIS

A

ABCNT • 3–25
ABORT • 5–52
Abort With Status • 5–54, 5–55
ABORTWS
  See Abort With Status
ACCESS • 5–53, 5–65
Access Paths Attention Message • 2–32, 2–51
Access_Path_Attn • 2–34
Accumulated Byte Count
  See ABCNT
ACP$GB_WINDOW • 3–8
ACP$READBLK • 3–24
ACP$WRITEBLK • 3–24
ACP QIO • 3–16
ACP_WINDOW • 3–8
Active I/O Requests • 4–40
Active Node • 1–3
Adapter Control Block • 3–40
ALLOCATE_HRB • 5–42
Allocation Class • 4–32, 5–1
AST_REC • 2–35
Attention Messages • 2–34
ATTN_MSG • 2–49
Attn_msg Routine • 2–34
Autoconfigure All • 2–26
Auto-Serving • 5–25
AVAILABLE • 5–53
Available Attention • 2–24

B

BACK
  See MSCP Routine BACK
Bad Block Replacement • 3–6, 4–13
BBR
  See Bad Block Replacement
BCNT • 3–24
BDT
  See Buffer Descriptor Table
BDTE
  See Buffer Descriptor Table Entry
Block Count Field • 3–8
Block Data Transfers • 1–4
  Mapping and Unmapping • 1–16
BOFF • 3–24
BOO$CONFIGURE • 2–51
BOO$CONNECT • 2–35
Boo$gl_dskdrv • 2–25
Boo$gl_prtdrv • 2–25
Buffer Descriptor • 1–26
Buffer Descriptor Table • 1–26
Buffer Descriptor Table Entry • 1–26
Buffered I/O Function Mask • 3–13
Buffer Handle • 1–26, 1–27
Buffer Name • 1–27
Buffer Ring • 1–29
Bugcheck • 4–59
Build_pdt • 3–34
Byte Offset Bit • 3–39

C

CALC_MVTIMEOUT • 4–50
Cancel I/O • 3–11
Cathedral Window • 3–8
CCB
  See Channel Control Block
CCB$B_AMOD • 3–3, 3–23
CCB$L_UCB • 3–3
CCB$L_WIND • 3–3
CDDB
  See Class Driver Data Block
CDDB$Q_CNTRLID • 5–13
CDDB$V_IMPEND • 4–3
CDDB$V_PATHMOVE • 4–19
CDDB$W_LOAD_AVAIL • 4–4
CDDB$W_STATUS • 4–3
CDL
  See Connection Descriptor List
CDRP
  See Class Driver Request Packet
CDRP$L_MSG_BUF • 3–37
CDT
  See Connection Descriptor Table

CDT$L_AUXSTRUC • 5–42, 5–83
Channelcnt • 3–2
Channel Control Block • 3–2
Channel Request Block • 2–16
  AUXSTRUC • 2–18
  Duetime • 2–17
  TOUTROUT field • 2–17
CI
  See Computer Interconnect
Class Driver • 1–1
Class Driver Data Block • 2–1
  Description • 2–11
Class Driver Input Dispatching Routine • 2–49
Class Driver Request Packet • 1–24, 3–10
CLU$GL_MSCP_CREDITS • 5–25
CLU$GL_MSCP_LOAD • 5–26
CLU$GL_MSCP_SERVE_ALL • 5–25
CLUSTRLOA
  See SYS$CLUSTER
CMD_STS • 5–41
Command Buffer • 1–16
Command Reference Number • 1–24
Command Ring • 1–16, 3–21, 3–37, 3–43
Computer Interconnect • 1–5
COMP_CTRL_DATA • 5–65
CONFIGMN • 2–35
Configure Process • 1–15, 1–20, 2–26, 2–35, 2–51
  Auto Serving of Disks • 2–35
Connection • 1–4
  Definition • 1–8
Connection Blocks
  See Connection Descriptor Table
Connection Descriptor List • 1–9
Connection Descriptor Table • 1–9
Connection Manager
  See VMS Connection Manager
Connection Walk • 4–37, 4–51, 4–56
Connectivity • 1–3
Connect Response • 1–20
Controller Based Shadow Set • 4–46
Controller Identifier • 5–15
Controller Online • 2–24
Controller Timeout Interval • 5–25
CRB
  See Channel Request Block
CRB$L_AUXSTRUC • 2–18
Create_device_entry • 3–34
Credit_stall • 4–4, 4–5
CTL$GL_CCBBASE • 3–2
CTL$GW_CHINDX • 3–4, 3–23

D

DAP
  See Determine Access Paths
DAP IRP/CDRP • 2–15
DAP_COUNT • 2–33
Datagrams • 1–3
Data Path • 3–40
Data Security Erase • 5–63
DDB
  See Device Data Block
DDT
  See Driver Dispatch Table
DDT$L_MNTVER • 4–49
DDT$L_START • 3–26
DELUA • 2–4
DEMNA • 2–4
Determine Access Paths • 2–15, 2–32, 5–69
DEUNA • 2–4
Device Data Block • 2–1, 2–5
Digital Storage Architecture • 1–1
Digital Storage Systems Interconnect • 1–5
Direct Memory Access • 3–19
Directory Lookup
  See SCS Directory Service
Disk Class Driver
  See DUDRIVER
Disk Device Naming Convention • 5–11
DISKSERVE BUGCHECK • 5–86
Disk Server Structure • 5–8
DMA
  See Direct Memory Access
DOUBLDEALO BUGCHECK • 5–86
DO_ACTION • 4–10
DO_DISK • 5–38
DO_ORIG_UCB • 2–29
Driver Dispatch Table • 3–2, 3–9
Driver Interrupt Service Routine • 2–16
DSA
  See Digital Storage Architecture
DSDRIVER • 1–2
DSE
  See Data Security Erase
DSRV
  See Disk Server Structure
DSRV$B_HOSTS • 5–19
DSRV$L_AVAIL • 5–26
DSRV$L_BUFFER_MIN • 5–24, 5–26, 5–44
DSRV$L_FREE_LIST • 5–21
DSRV$L_MEMW_BL
  See DSRV Memory Wait Queue
DSRV$L_SRVBUF • 5–26
DSRV$L_UNITS • 5–11, 5–43
DSRV$Q_CTRL_ID • 5–15
DSRV$W_LM_LOAD1 • 5–32

DSRV$W_LM_LOAD2 • 5–32
DSRV$W_LM_LOAD3 • 5–32
DSRV$W_LM_LOAD4 • 5–32
DSRV$W_LOAD_AVAIL • 5–32
DSRV$W_LOAD_CAPACITY • 5–30
DSRV$W_MEMW_CNT • 5–58
DSRV Memory Wait Queue • 5–44
DSRV Unit Table • 5–11
DSSI
  See Digital Storage Systems Interconnect
DU$CONNECT_ERR • 4–18
DU$CRESHAD_FDT • 2–45
DU$IDR • 3–31, 4–10
DU$INIT_HIRT • 2–40
DU$MSG_ERR_HNDLR • 4–13
DU$RE_SYNCH • 2–34, 4–3, 4–18, 4–29
DU$RE_SYNCH_PKT • 4–3
DU$SHADOW_GTCMD_UNIT • 4–5
DU$TMR
  See DUDRIVER Timeout Mechanism
Dual-Pathing • 4–32
Dual Ported Disk • 5–2
Dual Port Extension • 2–10
DUDRIVER • 1–2
  Timeout Mechanism • 4–1, 5–40
  Unit Polling • 2–43
DUETIME • 2–17
Duplicate Unit Attention Message • 2–51
Duplicate_Unit_Attn • 2–34
DUTU$BEGIN_CONN_WALK • 4–37, 4–51
DUTU$BEGIN_MNTVER • 4–40
DUTU$DODAP • 2–32, 2–48
DUTU$DRAIN_CDDB_CDRPQ • 4–23
DUTU$END_MNTVER • 4–41
DUTU$END_SINGLE_STREAM • 4–57
DUTU$LOCATE_UNIT • 4–37, 4–51
DUTU$LOOKUP_UCB • 2–46
DUTU$MOUNTVER • 4–49
DUTU$NEW_UNIT • 2–29, 2–45
DUTU$POLL_FOR_UNITS • 2–29, 2–43
DUTU$RESTART_NEXT_CDRP • 4–42, 4–57
DUTU$REVALIDATE • 4–20, 4–30
DUTU$SETUP_DUAL_PATH • 2–32, 2–45
DUTU$TERMINATE_PENDING • 4–46
DUTUMAC • 4–11
DU_BEGIN_IVCMD • 4–14
DU_CONTROLLER_INIT • 2–29, 2–39
DU_RESTARTIO • 4–14
DU_STARTIO • 3–26
DW780 Unibus Adapter Technical Description • 3–40
Dynamic Load Balancing • 5–31

E

EMB$K_CLTRES_IMTMO • 4–3
Enable Set Write Protect • 5–68
End Message
  See MSCP End Message
End Mount Verification Routine
  See DUTU$END_MNTVER
Envelope Address • 3–44
ERASE • 5–53
ERL$LOGSTATUS • 4–16
Errorlog Entry
  Determining conditions for • 4–13
  DU$MSG_ERR_HNDLR • 4–13
  EMB$C_DUPUN • 2–51
  EMB$C_INVATT • 2–50
Error Log Extension • 2–9
Error Logging • 4–16
Error Status • 4–7
Ethernet • 1–5
EXDRIVER • 2–4
EXE$ALLOCIRP • 3–23
EXE$CLUTRANIO • 4–30
EXE$FINISHIOC • 3–18
EXE$INSIOQ • 3–26, 3–33
EXE$INSIOQC • 5–38
EXE$MOUNTVER • 4–14, 4–29, 4–30
EXE$QIOACPPKT • 3–26
EXE$QIODRVPKT • 3–26
EXE$QIORETURN • 3–26
EXE$READLOCK • 3–24
EXE$TIMEOUT • 2–17
EXE$WRITELOCK • 3–24
EXIT_ATTN_MSG • 2–50

F

FDT
  See Function Decision Table
FIND_UQB • 5–43
FLUSH • 5–67
FORKLOCK • 3–26
Fork Lock Field • 5–42
Fork Process • 1–15, 1–24
Fork Spinlock • 3–26
FOUND_PROC • 2–51
FPC$ALLOCMSG • 3–27, 3–36
FPC$DEALLOMSG • 3–50
FPC$SNDCNTMSG • 3–29, 3–45
Function Decision Table • 3–9

G

Get Command Status • 5–52
Get Unit Status • 2–29
  Next Unit • 5–55

H

HANDLE_INT • 3–30
Hardware Write Protect • 5–48
Hello Packet • 2–4
Hierarchical Storage Controller • 5–1
HIRT
  See Host Initiated Replacement Table
Host Based Shadow Set • 4–46
Host Clear • 4–19, 4–24, 5–52
Host Initiated Replacement Table • 2–40, 4–23
Host Number • 5–19, 5–28
Host Queue Block • 5–6
Host Request Block • 5–5
Host Unit Load Block • 5–8
  Operation Count • 5–8
HQB
  See Host Queue Block
HQB$Q_TIME • 5–55
HRB
  See Host Request Block
HRB$B_LBUFF • 5–38
HRB$K_SNDAT_WAIT • 5–47, 5–49
HRB$K_ST_MAP_WAIT • 5–45
HRB$K_ST_MSG_WAIT • 5–51
HRB$K_ST_SEQ_WAIT • 5–44
HRB$K_ST_SNDMS_WAIT • 5–51
HRB$L_BUFADR • 5–44
HRB$L_BUFLEN • 5–44
HRB$L_HQB • 5–42
HRB$L_RESPC • 5–38
HRB$W_STATE • 5–44
HSC
  See Hierarchical Storage Controller
HULB
  See Host Unit Load Block
  Operation Count • 5–39, 5–40, 5–46
Hustvedt
  See RAD Hustvedt

I

I/O Channel Number • 3–1
I/O Postprocessing • 5–46
I/O Postprocessing Routine
  Back • 5–45
I/O Request Packet • 1–22, 3–9
I/O Status Block • 3–9
IDR
IDR (Cont.)
  See Class Driver Input Dispatching Routine
IDREC • 2–4
IF_IVCMD • 4–14
IF_MSCP • 4–10
IMMEDIATE • 5–54
Immediate Class Commands • 4–3
Immediate Commands • 5–52
INIT_UDA_BUFFERS • 3–44
Ini_bootdevic • 2–24
Integrated Storage Element • 5–1
INVALID_STS • 4–29
IO$PACK_ACK • 4–27
IO$_ACCESS • 3–5, 3–16
IO$_AVAILABLE • 4–15
IO$_CRESHAD • 2–45
IO$_DSE • 5–63
IO$_NOP • 4–15
IO$_PACKACK • 4–15
IO$_READPBLK • 5–38
IOC$ALTREQCOM • 4–29
  JMP • 4–58
  JSB • 4–58
IOC$COPY_UCB • 2–48
IOC$CVTLOGPHY • 3–25, 5–45
IOC$CVT_DEVNAM • 5–28
IOC$FFCHAN • 3–4
IOC$GL_CRBTMOUT • 2–17, 4–1
IOC$GL_DEVLIST • 2–7
IOC$GL_DU_CDDB • 4–30
IOC$GQ_POSTIQ • 3–32
IOC$GW_MVTIMEOUT • 4–39
IOC$INITDRV • 2–25
IOC$INITIATE • 3–26, 4–55
IOC$IOPOST • 5–40, 5–46
IOC$LUBAUDAMAP • 3–40
IOC$MAPVBLK • 3–8, 3–25
IOC$REQMAPUDA • 3–40
IOC$SEARCH • 3–4
IOSB
  See I/O Status Block
IPL$_IOPOST • 5–46, 5–50
IRP
  See I/O Request Packet
IRP$L_MEDIA • 3–25
IRP$L_SEGVBN • 3–25, 3–33
IRP$W_STS • 3–25
IRP/CDRP Pair • 3–10
ISE
  See Integrated Storage Element
ISR
  See Driver Interrupt Service Routine

K

KDM70 Local Port • 3–34

L

LBN
  See Logical Block Number
LCONID
  See Local Connection Identifier
LINK_NEW_UCB • 2–48
Listening CDT • 1–13
LM_INIT • 5–31
LM_INIT_CAPACITY • 5–30
Load Availability • 5–31
Load Available • 4–4
Load Balance • 4–37, 4–51, 4–55, 5–8
Load Capacity • 5–2, 5–8, 5–30
Load Monitoring Thread • 5–25, 5–31
Load Monitor Initialization Routine • 5–32
Load Monitor Interval • 5–31
Load Threshold • 5–32
LOAD_BALANCE Routine • 5–32
LOAD_MONITOR Routine • 5–31
Local Connection Identifier • 1–9
  Definition • 1–9
Lock Manager
  See VMS Lock Manager
Logical Block Number • 3–6
Logical Blocks • 3–6
Logical Controller • 5–3

M

Maintainence Commands • 5–37
Make Connection • 4–20
MAKE_CONNECTION • 2–29
MAP • 5–45, 5–48
Mapping Registers • 3–38
MAP_IRP • 3–28
Massbus • 5–2
Mass Storage Control Protocol • 1–1, 5–1
Master File Directory • 3–4
Message Buffer • 1–16
Message Free Queue • 1–29
Messages • 1–4
MFD
  See Master File Directory
Mountver • 4–27
Mount Verification • 4–20, 4–26
  Connection Manager Induced • 4–30
Mount Verification IRP • 4–43
Mount Verification Timeout Period
  See MVTIMEOUT
MSCP
  See Mass Storage Control Protocol
MSCP$ADDUNIT • 5–27
MSCP$K_CM_EMULA • 4–19
MSCP$K_OP_READ • 3–28
MSCP$K_OP_WRITE • 3–28
MSCP$K_SLUN_RSVP • 5–43
MSCP$K_ST_AVLBL • 4–56, 5–63
MSCP$K_ST_ICMD • 5–37
MSCP$K_ST_SUCC • 4–56
MSCP$K_ST_WRTPR • 5–65
MSCP$L_CMD_REF • 3–31
MSCP$TMR • 5–32
MSCP$V_EF_ERLOG • 4–16
MSCP$V_MD_NXUNT • 5–55
MSCP.EXE
  See VMS based MSCP Disk Server
MSCP Class Field • 5–36
MSCP Commands
  Immediate • 5–52
  Nonsequential Buffered • 5–36
  NonSequential Nonbuffered • 5–52
  Sequential • 5–52
MSCP Disk Serving
  Automatic • 5–2
  Selective • 5–2
MSCP End Message • 1–23, 4–6
MSCP Error Status
  Error Codes • 4–11
  Major Status Code • 4–7
  Sub-Code • 4–7
  VMS error to MSCP error translation • 5–72
MSCP Extension • 2–10
MSCP Reset • 4–19
MSCP Routine BACK • 5–38
MSCP Routine READ • 5–37
MSCP Routine WRITE • 5–37
MSCPSERV BUGCHECK • 5–85
MSCP Server • 1–1
  Dynamic Load Balancing • 5–31
  Load Availability • 5–31
  Load Capacity • 5–30
  Static Load Balancing • 5–30, 5–31
MSCP Start • 4–19
MSCP Unique Identifiers • 5–13
MSCP Unit Identifier • 5–11
MSCP Unit Number • 5–11
MSCP_BUFFER • 5–23
MSCP_CREDIT • 5–25
MSCP_LOAD • 5–1
MSCP_SERVE_ALL • 5–25
MSGLENGTH_ARRAY • 3–34
MSG_IN • 5–35, 5–37
MVTIMEOUT • 4–37, 4–39
  Default Value • 4–39

N

Network Interconnect • 1–5
New I/O Requests • 4–40
NI
  See Network Interconnect
NISCS_LOAD_PEA0 • 2–25
NONSEQB • 5–37, 5–42
NonSequential Buffered Commands • 5–36, 5–37, 5–42
NonSequential Nonbuffered Commands • 5–52

O

OBCNT • 3–24
Odd Byte Count Error • 4–15
ONLINE • 5–53
Online Command • 5–20
Original Byte Count
  See OBCNT

P

PACKACK_UNKNO_INOPR • 4–38
Packack_Volume • 4–35
PADRIVER • 1–5
Page Frame Number • 3–38
Page Table Entry • 1–26
Passive Node • 1–7
Path Block • 1–7
Pathmove • 2–41
Path Move • 4–31, 4–55
PB
  See Path Block
PDT$B_CRINGCNT • 3–46
PDT$B_CRINGINX • 3–43
PDT$B_RPOLLINX • 3–47
PDT$B_RRINGINX • 3–47
PDT$L_BUFARY • 3–34
PDT$L_CMDRING • 3–43
PDT$L_CRCONTENT • 3–34
PDT$L_NO_BUFFS • 3–34
PDT$L_PU_BUFQFL • 3–37
PDT$L_PU_CDTARY • 3–49
PDT$L_PU_FQBL • 3–47
PDT$L_PU_FQFL • 3–34
PDT$L_PU_PORTCHAR • 3–46
PDT$L_PU_SNDQFL • 3–46
PDT$L_RINGSIZE • 3–43
PDT$L_RRCONTENT • 3–34
PDT$L_RSPRING • 3–47
PDT$L_SNDCNTMSG • 3–28
PDT$L_UDAB_LEN • 3–34
PE$INT • 3–21
PEM
  See Port Emulator
Pending I/O Queue • 4–40
Pending IRP Queue • 4–40
PERFORM_VALIDATE • 4–35
Permanent IRP/CDRP • 2–15
PFN
  See Page Frame Number
Physical Blocks • 3–5
Physical Interconnect • 1–11
  Description • 1–17
PI
  See Physical Interconnect
PIDRIVER • 1–5
Ping Pong • 4–38
POLL_RSPRING • 3–47
Port • 1–4
  Definition • 1–4
Port Driver • 1–5
Port Emulator • 1–5, 2–4
Port to Port Driver • 1–11
  Description • 1–17
PPD
  See Port to Port Driver
Preferred Path • 4–37, 4–51
Protocol • 1–11
Pseudo Map Registers • 3–42
PTE
  See Page Table Entry
PU$INT
  See Pudriver Interrupt Service Routine
PUDRIVER • 1–6, 3–33
Pudriver Interrupt Service Routine • 3–47

Q

Q-bus • 3–42

R

RAD Hustvedt • 5–19, B–57
RBN
  See Replacement Block Number
RCONID
  See Remote Connection Identifier
RDT
  See Request Descriptor Table
RDTE
  See Request Descriptor Table Entry
READ
  See MSCP Routine READ
RECORD_ONLINE • 4–57
RECORD_UNIT_STATUS • 4–57
RECYCL_MSG_BUF • 5–51
Register Dump • 3–11
Relative Volume Table • 3–8

Remote Connection Identifier • 1–9
Definition • 1–9
REPLACE • 5–53, 5–67
Replacement Block Number • 3–6
Replacement Blocks • 3–6
Request Descriptor Table • 1–24
Request Descriptor Table Entry • 1–24
REQUEST_DATA Macro • 5–40
Resource Wait Count • 4–40
Response Descriptor Table • 4–29
Response Identifier • 1–24
Service • 1–24
Response Queue • 2–4
Response Ring • 1–16, 3–21, 3–47
REVAL_HBS_MEMBER • 4–31, 4–46
Ringexp_array • 3–34
Root Volume • 3–4
Routine SCAN_ALL_DEVICES • 2–35
RSPID
See Response Identifier
RVT
See Relative Volume Table
RWAITCNT • 3–27, 4–23
See Resource Wait Count

S

SB
See System Block
SB$B_ENBMSK • 1–14
SBI
See Synchronous Backplane Interconnect
SCA
See Systems Communications Architecture
SCAN_RSPID_WAIT • 4–23
SCS
See Systems Communication Services
SCS$C_USE_ALTERNATE_PORT • 4–19, 5–83
SCS$DIRECTORY
See SCS Directory Service
SCS$DIR_LOOKUP
See SCS Process Poller Service
SCS$FPC_MAP • 5–45
SCS$FPC_MAPIRP • 3–28
SCS$FPC_UNMAP • 3–32
SCS$GL_MSCP • 5–9
SCS$GQ_CONFIG • 1–8
See System Block Configuration Listhead
SCS$POLL_MBX • 1–15
SCS$POLL_PROC • 1–14
SCS$UNSTALLUCB • 4–41
SCSCI$FORK • 3–30
SCSCI$PROCESS_RSP_PPD • 3–30
SCSCI$SNDMSG • 3–29
SCS Directory Entry • 1–13
SCS Directory Service • 1–13
SCS Flow Control • 1–28
SCSLOA
See SYS$SCS
SCS Process Name Block • 1–14
SCS Process Poller Service • 1–14
SCS Process Polling Block • 1–14, 2–52
SDI
See Standard Disk Interface
SDIR
See SCS Directory Entry
Secondary Offset Into A Buffer • 1–28
Segment • 3–9
Send Credits • 1–28
SEND_DATA Macro • 5–39
SEND_END • 5–51
SEND_MSCP_MSG • 4–10
Sequence Number Order • 4–41
Sequential Commands • 5–52
Server Local Unit Number • 2–43, 5–7, 5–8, 5–27, 5–43
Server Mount Verification Routine • 4–49
Set Controller Characteristics • 4–18, 5–61
SET UNIT CHARACTERISTICS • 5–53
SGN$GW_PCHANCNT • 3–2
SLUN
See Server Local Unit Number
SNDDAT • 5–39
Software Interrupt
IPL$_IOPOST • 5–46, 5–50
Software Invalid • 4–39
Software Write Protect • 5–48, 5–68
Special Credit • 1–29
Special Dudriver Extension • 2–10
Spindown • 5–68
SPL$C_SCS • 5–42
Split I/O • 3–25
SPNB
See SCS Process Name Block
SPPB
See SCS Process Polling Block
SRVBUF$L_SIZE • 5–26
SS$_CTRLERR • 4–59
SS$_VOLINV • 4–42
STACONFIG
See Standalone Configure Process
See Stand Alone Configure Process
Standalone Configure Process • 5–1
Stand Alone Configure Process • 2–25
Standard Disk Extension • 2–10
Standard Disk Interface • 4–34
Start/Stack/Ack dialogue • 2–4
Start I/O • 3–11
Startio Routine • 1–24
START_NOP • 4–15
Start_Packack • 4–56
Retry Attempts • 4–56
START_PACKACK • 4–15

START_READBLK • 3–28
START_WRITEBLK • 3–28
STATE_INVALID • 5–42
Static Load Balancing • 5–30, 5–31
SVAPTE
  See System Virtual Address Page Table Entry
$SYNCH • 3–18
Synchronous Backplane Interconnect • 3–38
SYS$ASSIGN • 3–1
SYS$CLUSTER
  SYSAP • 1–12
SYS$GL_BOOTDDB • 2–8
SYS$SCS • 1–12
SYSAP
  See System Application
Sysgen
  MSCP load command • 5–1
Sysgen Parameter
  ACP_WINDOW • 3–8
  CHANNELCNT • 3–2
  MSCP_BUFFER • 5–23
  MSCP_CREDIT • 5–25
  MSCP_LOAD • 5–1, 5–30
  MSCP_SERVE_ALL • 2–35, 5–2, 5–25
  MVTIMEOUT • 4–39
  NISCS_LOAD_PEA0 • 2–25
System Application
  Definition • 1–12
System Block • 1–7
System Block Configuration Listhead • 2–3
System Block Enable Mask • 1–14
Systems Communications Architecture
  definition • 1–1
Systems Communication Services
  Accept • 1–12
  Connect • 1–12
  definition • 1–1
  Disconnect • 1–13
  Listen • 1–13
  Port Dependent Layer • 1–12
  Port Independent Layer • 1–12
  Reject • 1–12
System Virtual Address Page Table Entry • 1–26

T

Tape Class Driver
  See TUDRIVER
Tape Mass Storage Control Protocol • 1–3
Timer Queue Entry • 4–1
TMSCP
  See Tape Mass Storage Control Protocol
TQE
  See Timer Queue Entry
Transfer Buffers • 5–21
Transfer Segments • 3–9
TRANSFER_MSCP_ERROR • 4–10
TUDRIVER • 1–3

U

UCB
  See Unit Control Block
UCB$L_CDDB
  See UCB pointer to CDDB
UCB$L_IOQFL • 4–23, 4–40
UCB$L_PREF_CDDB • 4–37, 4–51
UCB$V_MSCP_MNTVERIP • 4–40
UCB$W_ERRCNT • 4–16
UCB Extensions • 2–9
  Dual Port • 2–10
  Error Log • 2–9
  MSCP • 2–10
  Special Dudriver • 2–10
  Standard Disk • 2–10
UCB pointer to CDDB • 2–12
UDA$K_MAX_RINGSIZE • 3–34
UDAB$L_DESCRIP • 3–44
UDAB$T_TEXT • 3–34
UDAB buffer • 3–34
Unibus • 3–37
Unique Device Number • 5–16
Unit Available Attention Message • 2–50
Unit Control Block
  Creation • 2–8
  Definition • 2–2
  Error Count • 4–13
Unit Identifier • 5–43
Unit Initialization • 3–11
Unit Polling • 2–43
Unit_available_attn • 2–34
UNMAP
  See SCS$FPC_UNMAP
UQB$B_ONLINE • 5–20
UQB$L_BLOCKED_BL • 5–75
UQB$L_BLOCKED_FL • 5–75
UQB$W_OLD_UNIT • 5–13
UQB$W_SLUN • 5–13

V

Validate_Volume • 4–35
Valid I/O Function Mask • 3–13
VBN
  See Virtual Block Number
VC
  See Virtual Circuit
Version Specific Constants
  AST_REC timer • 2–35
  CDRP Restart Retry Count • 4–59
  CONNECT_DELTA • 2–41, 4–20
  Controller Identifier • 5–13

Version Specific Constants (Cont.)
Credit Stall Timeout • 4–4, 4–5
DAP_COUNT • 2–33
Default Buffers • 3–34
Dynamic Load Balancing • 5–31
EMB$C_ACPTH • 2–51
Example • 1–7
Host Bitmap • 5–28
Host Timeout • 5–28
HOST_TIMEOUT • 2–42
Load Monitor Interval • 5–31, 5–32
Local Controller Default Buffer Count • 3–35
Local Controller Default Ringsize • 3–43, 3–47
Mount Verification Timeout • 4–39, 4–50
MSCP_CREDITS • 5–25
Ping Pong Retry Limit • 4–38
Stalled Request Buffers • 3–34
Start Packack Retry Limit • 4–56
Tape Serving • 1–3
UDA$K_MAX_RINGSIZE • 3–34
Unit Identifier • 5–19
WCB Limitations • 3–6
Virtual Block Number • 3–6
Virtual Blocks • 3–5
Virtual Circuit • 1–4
Definition • 1–7
VMS based MSCP Disk Server • 2–6, 5–1
VMS based MSCP server
Message Input Routine • 5–35
START • 5–26
VMS based MSCP server Routines
ACCESS • 5–65
ADD • 5–27
COMP_CTRL_DATA • 5–65
ERASE • 5–65
FLUSH • 5–67
LISTN • 5–28
REPLACE • 5–65, 5–67
VMS Connection Manager • 1–3
SYSAP • 1–12
VMS Crb Timeout Auxilary Structure
See AUXSTRUC
VMS Crb Timeout Routine
See TOUTROUT
VMS Device Timeout Check
See EXE$TIMEOUT
VMS Lock Manager • 1–3
SYSAP • 1–12
Volume Set • 3–7

W

$WAITFR • 3–18
WCB
See Window Control Block
WCB$L_STVBN • 3–7
WCB$W_NMAP • 3–8
Window Control Block • 3–3, 3–5
Block Count Field • 3–8
limitations • 3–6
Mapping • 3–7
Window Turn • 3–8, 3–26
WRITE • 5–39
See MSCP Routine WRITE
Write Attention AST • 2–52

X

XFER_REPLACE • 4–29