MicroPower/Pascal I/O Services Manual

Order No. AA-FQ15C-TK

digital software

MicroPower/Pascal I/O Services Manual

Order No. AA-FQ15C-TK

June 1987

This manual contains the I/O services information required for designing and developing MicroPower/Pascal microcomputer application programs. I/O services include file system services, task-to-task communication, and device I/O. A guide to writing device drivers is provided for designing and developing nonstandard device drivers.

Operating System and Version: Micro/RSX Version 3.0
RSX-11M Version 4.2
RSX-11M-PLUS Version 3.0
RT-11 Version 5.2
VAX/VMS Version 4.0

Software Version: MicroPower/Pascal-Micro/RSX Version 2.4
MicroPower/Pascal-RSX Version 2.4
MicroPower/Pascal-RT Version 2.4
MicroPower/Pascal-VMS Version 2.4

Digital Equipment Corporation Maynard, Massachusetts

First Printing, June 1985
Updated, April 1986
Revised, October 1986
Revised, June 1987

The 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	PDP	UNIBUS
DECmate	P/OS	VAX
DECUS	Professional	VMS
DECwriter	Rainbow	VT
DIBOL	RSTS	Work Processor
MASSBUS	RSX	digital
MicroPower/Pascal	RT	

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

Preface xix

Chapter 1 Introduction to MicroPower/Pascal Input/Output

1.1 I/O System Architecture 1-3
1.2 Performing I/O 1-7
1.3 Request/Reply Packet Interface 1-8
1.3.1 Request Queue Names 1-9
1.3.2 I/O Request and Reply Packets 1-11

Chapter 2 Ancillary Control Process

2.1 ACP Features and Capabilities 2-1
2.2 Accessing the ACP for File I/O 2-2
2.3 Pascal File System Interface 2-3
2.4 Request/Reply Packet Interface 2-4
2.4.1 Physical Read and Write Functions 2-4
2.4.2 Logical Read and Write Functions 2-5
2.4.3 Set Characteristics Function 2-5
2.4.4 Get Characteristics Function 2-5
2.4.5 Lookup and Enter Functions 2-6
2.4.6 Rename, Delete, Protect, and Unprotect Functions 2-7
2.4.7 Close and Purge Functions 2-7
2.5 Status Codes 2-8
2.6 ACP Prefix File 2-9
2.7 Application Note: Device-Name Parsing 2-10
2.8 FALACP 2-11

Chapter 3 Asynchronous Serial Line (Terminal) Driver

3.1 TT Driver Features and Capabilities 3-1
3.2 Performing Asynchronous Serial I/O 3-2
3.3 Pascal I/O Procedure Interface 3-3
3.4 Request/Reply Packet Interface 3-5
3.4.1 Read Functions 3-7
3.4.2 Write Functions 3-8
3.4.3 Get and Set Characteristics Functions 3-9
3.4.4 Set Modem Semaphore Function 3-14
3.4.5 Stop Request 3-15
3.5 Status Codes 3-15
3.6 TT Driver Prefix File 3-16
3.7 Application Note: Hardware Buffering 3-20

Chapter 4 Disk-Class Device Drivers

4.1 Disk Driver Features and Capabilities 4-2
4.2 Performing Disk I/O 4-3
4.3 Pascal I/O Procedure Interface 4-5
4.4 Request/Reply Packet Interface 4-7
4.4.1 RL01/2 (DL) Functions 4-10
4.4.1.1 DL Logical Read and Write 4-10
4.4.1.2 DL Physical Read and Write 4-11
4.4.1.3 DL Get Characteristics 4-12
4.4.2 RX02 (DY) Functions 4-13
4.4.2.1 DY Logical Read and Write 4-13
4.4.2.2 DY Physical Read and Write 4-14
4.4.2.3 DY Format Subfunctions of Physical Write 4-15
4.4.2.4 DY Get Characteristics 4-16
4.4.3 MSCP (DU) Functions 4-17
4.4.3.1 DU Logical Read and Write 4-17
4.4.3.2 DU Get Characteristics 4-18
4.4.4 Extended Disk (XD) Functions 4-18
4.4.4.1 XD Logical Read and Write 4-18
4.4.4.2 XD Get Characteristics 4-19
4.4.5 TU58 (DD) Functions 4-20
4.4.5.1 DD Logical Read and Write 4-20
4.4.5.2 DD Physical Read and Write 4-21
4.4.5.3 DD Get Characteristics 4-22

4.4.6 Virtual Memory (VM) Functions 4-23
4.4.6.1 VM Logical Read and Write 4-23
4.4.6.2 VM Get Characteristics 4-23
4.5 Status Codes 4-24
4.6 Extended Error Information 4-26
4.7 Disk Driver Prefix Files 4-26
4.8 Extended Disk Driver Source Excerpt 4-30

Chapter 5 TMSCP Tape Driver

5.1 MU Driver Features and Capabilities 5-1
5.2 Performing TMSCP Tape I/O 5-2
5.3 Pascal Support Routine Interface 5-4
5.3.1 READ_TAPE 5-5
5.3.2 WRITE_TAPE 5-5
5.3.3 REPOSITION_TAPE 5-6
5.3.4 WRITE_TAPE_MARK 5-7
5.3.5 REWIND_TAPE 5-7
5.4 Pascal I/O Procedure Interface 5-8
5.5 Request/Reply Packet Interface 5-9
5.5.1 Read and Write Functions 5-11
5.5.2 Get Characteristics Function 5-12
5.5.3 Reposition Tape Function 5-12
5.5.4 Write Tape Mark Function 5-13
5.5.5 Rewind Tape Function 5-13
5.6 Status Codes 5-14
5.7 MU Driver Prefix File 5-15

Chapter 6 Parallel Line Drivers

6.1 Parallel Line Driver Features and Capabilities 6-2
6.2 Performing Parallel I/O 6-3
6.3 Pascal I/O Procedure Interface 6-5
6.4 Pascal Support Routines 6-7
6.4.1 SBC-11/21 PIO Support Routines 6-8
6.4.1.1 SET_PIO_MODE 6-8
6.4.1.2 WRITE_PIO 6-8
6.4.1.3 READ_PIO 6-9

6.4.2 KXT11-CA/KXJ11-CA PIO and Counter/Timer Support Routines 6-9
6.4.2.1 YK_PORT_READ 6-10
6.4.2.2 YK_PORT_WRITE 6-11
6.4.2.3 YK_SET_PATTERN 6-12
6.4.2.4 KXT11-CA/KXJ11-CA PIO DMA Process 6-15
6.4.2.5 YK_SET_TIMER 6-19
6.4.2.6 YK_READ_TIMER 6-20
6.4.2.7 YK_CLEAR_TIMER 6-21
6.4.2.8 Using Timer/Counters to Count External Pulses 6-21
6.4.2.9 Linking Two Timer/Counters as 32-Bit Counter 6-24
6.5 Request/Reply Packet Interface 6-25
6.5.1 DRV11-J (XA) Functions 6-29
6.5.1.1 XA Read and Write 6-29
6.5.1.2 XA Get Characteristics 6-30
6.5.1.3 XA Enable 6-30
6.5.1.4 XA Disable 6-31
6.5.2 DRV11 (YA) Functions 6-31
6.5.2.1 YA Read and Write 6-31
6.5.2.2 YA Get Characteristics 6-32
6.5.3 DRV11-B (YB) Functions 6-33
6.5.3.1 YB Read and Write 6-33
6.5.3.2 YB Set Characteristics 6-35
6.5.3.3 YB Get Characteristics 6-36
6.5.4 SBC-11/21 PIO (YF) Functions 6-36
6.5.4.1 YF Read and Write 6-36
6.5.4.2 YF Get Characteristics 6-37
6.5.5 KXT11-CA/KXJ11-CA PIO (YK) Functions 6-38
6.5.5.1 YK Read 6-38
6.5.5.2 YK Write 6-38
6.5.5.3 YK Get Characteristics 6-39
6.5.5.4 YK Set Pattern 6-40
6.5.5.5 YK DMA Read, Write, and Complete 6-41
6.5.5.6 YK Set Timer 6-42
6.5.5.7 YK Clear Timer 6-43
6.5.5.8 YK Read Timer 6-43
6.6 Status Codes 6-43
6.7 Extended Error Information 6-44
6.8 Parallel Line Driver Prefix Files 6-45
6.8.1 XA Prefix File 6-45
6.8.2 YA Prefix File 6-46

6.8.3 YB Prefix File 6-47
6.8.4 YF Prefix File 6-48
6.8.5 YK Prefix File 6-50

Chapter 7 Analog-to-Digital Converter Driver

7.1 Driver Features and Capabilities 7-1
7.2 Performing Analog-to-Digital Conversions 7-2
7.3 Pascal I/O Procedure Interface 7-3
7.4 Pascal Support Routine Interface 7-4
7.4.1 SET_ANALOG_MODE 7-5
7.4.2 READ_ANALOG_SIGNAL 7-7
7.4.3 WRITE_ANALOG_WAIT 7-8
7.5 Request/Reply Packet Interface 7-8
7.5.1 Set Characteristics (Configure Device) Function 7-11
7.5.2 Read Logical (Read Converted Data) Function 7-13
7.5.3 Get Characteristics Function 7-14
7.6 Status Codes 7-15
7.7 AD Driver Prefix File 7-15

Chapter 8 Real-Time Clock Driver

8.1 KW Driver Features and Capabilities 8-1
8.2 Performing Real-Time Clock I/O 8-2
8.3 Pascal Support Routine Interface 8-3
8.3.1 READ_COUNTS_WAIT 8-3
8.3.2 READ_COUNTS_SIGNAL 8-6
8.3.3 START_RTCLOCK 8-8
8.3.4 STOP_RTCLOCK 8-10
8.4 Request/Reply Packet Interface 8-10
8.4.1 Read Physical Function 8-13
8.4.2 Enable Clock Function 8-15
8.4.3 Disable Clock Function 8-17
8.4.4 Get Characteristics Function 8-17
8.5 Status Codes 8-18
8.6 KW Driver Prefix File 8-18

Chapter 9 Peripheral Processor DMA Driver

9.1 QD Driver Features and Capabilities 9-1
9.2 Performing KXT11-CA/KXJ11-CA DMA I/O 9-2
9.3 Pascal Support Routine Interface 9-3
9.3.1 $DMA_TRANSFER 9-4
9.3.2 $DMA_SEARCH 9-6
9.3.3 $DMA_SEARCH_TRANSFER 9-7
9.3.4 KXT11-CA/KXJ11-CA PIO with DMA 9-8
9.3.5 KXT11-CA/KXJ11-CA I/O Using SLU2A or SLU2B with DMA 9-8
9.3.6 $DMA_GET_STATUS 9-9
9.3.7 $DMA_ALLOCATE 9-11
9.3.8 $DMA_DEALLOCATE 9-11
9.3.9 KXT11-CA/KXJ11-CA DMA Sample Program 9-11
9.4 Request/Reply Packet Interface 9-14
9.4.1 Read and Write Functions 9-16
9.4.2 KXT11-CA/KXJ11-CA PIO DMA Process 9-21
9.4.3 KXT11-CA/KXJ11-CA I/O Using SLU2A or SLU2B with DMA 9-22
9.4.4 Get Characteristics Function 9-22
9.4.5 Channel Allocation and Deallocation 9-24
9.5 Status Codes 9-25
9.6 QD Driver Prefix File 9-26

Chapter 10 Instrument Bus Driver

10.1 Instrument Bus Features and Capabilities 10-1
10.2 Driver Features and Capabilities 10-3
10.3 Performing Instrument Bus I/O 10-4
10.4 Pascal Support Routine Interface 10-5
10.4.1 READ_IEQ 10-6
10.4.2 WRITE_IEQ 10-7
10.4.3 SET_STATE 10-8
10.4.4 WRITE_EOI_IEQ 10-9
10.4.5 IEQ_COMMAND 10-10
10.4.6 IEQ_SERIAL 10-11
10.4.7 IEQ_PARALLEL_POLL 10-12
10.4.8 IEQ_PARALLEL_LOAD 10-13
10.4.9 IEQ_PARALLEL_CONFIG 10-13
10.4.10 IEQ_AUX_COMMAND 10-14
10.4.11 IEQ_REQ_SERVICE 10-15
10.4.12 IEQ_CONTROL_GTS 10-16
10.4.13 IEQ_PASS_CONTROL 10-17

10.4.14 SET_INT_MASK 10-17
10.4.15 REC_IEQ_EVENT 10-18
10.5 Request/Reply Packet Interface 10-20
10.5.1 Read Logical Function 10-23
10.5.2 Write and Write with EOI Termination Functions 10-24
10.5.3 Get Characteristics (Sense State) Function 10-25
10.5.4 Set Characteristics (Set State) Function 10-25
10.5.5 Write IEEE Remote Messages Function 10-27
10.5.6 Serial Poll Functions 10-28
10.5.7 Parallel Poll Function 10-29
10.5.8 Load Parallel Poll Register Function 10-29
10.5.9 Parallel Poll Configure Function 10-30
10.5.10 Auxilary Command Function 10-31
10.5.11 Request Service Function 10-32
10.5.12 Get Control Function 10-32
10.5.13 Go to Standby Function 10-33
10.5.14 Pass Control Function 10-33
10.5.15 Set Event Mask Function 10-34
10.5.16 Wait for Event and Recognize Event Functions 10-35
10.6 Status Codes 10-37
10.7 Extended Error Information 10-38
10.8 XE Driver Prefix File 10-38

Chapter 11 Network Service Process

11.1 NSP Features and Capabilities 11-1
11.2 Accessing the NSP for Task-to-Task Communication 11-2
11.3 Pascal File System Interface 11-4
11.4 NSP Set and Get Characteristics Functions 11-4
11.4.1 Set Characteristics to $SECTL Queue Semaphore 11-4
11.4.2 Get Characteristics to $SECTL Queue Semaphore 11-5
11.4.3 Get Characteristics to File Variable 11-6
11.5 Status Codes 11-6
11.6 NSP Prefix File 11-8
11.7 Sample Programs 11-11
11.7.1 Transferring Data Between Two MicroPower/Pascal Nodes 11-11
11.7.2 Transferring Data Between MicroPower/Pascal and VAX/VMS Nodes 11-13
11.7.3 Determining and Setting the Local Node Number 11-15

Chapter 12 Asynchronous DDCMP Driver

12.1 CS Driver Features and Capabilities 12-2
12.2 Performing Asynchronous DDCMP I/O 12-3
12.3 Pascal I/O Procedure Interface 12-6
12.4 Request/Reply Packet Interface 12-8
12.4.1 Enable Protocol and Disable Protocol Functions 12-11
12.4.2 Read and Write Functions 12-11
12.4.3 Get Characteristics Function 12-12
12.5 Status Codes 12-13
12.6 CS Driver Prefix File 12-13

Chapter 13 Communication Drivers

13.1 Communication Driver Features and Capabilities 13-2
13.1.1 Ethernet Communication 13-3
13.1.2 Synchronous Point-to-Point Communication 13-4
13.1.3 Peripheral Processor Two-Port RAM Communication 13-5
13.2 Performing Communication Device I/O 13-6
13.3 Pascal I/O Procedure Interface 13-10
13.4 Request/Reply Packet Interface 13-11
13.4.1 DEQNA (QN) Functions 13-15
13.4.1.1 QN Enable Portal 13-15
13.4.1.2 QN Read and Write 13-16
13.4.1.3 QN Get Characteristics 13-18
13.4.1.4 QN Disable Portal 13-18
13.4.2 DPV11 and KXT11-CA/KXJ11-CA Synchronous Communication (XP and XS) Functions 13-19
13.4.2.1 XP or XS Enable and Disable 13-19
13.4.2.2 XP or XS Read and Write 13-19
13.4.2.3 XP or XS Get Characteristics 13-20
13.4.2.4 XP or XS Stop 13-21
13.4.2.5 XP or XS Set Modem Semaphore 13-21
13.4.3 KXT11-CA/KXJ11-CA Two-Port RAM (KX and KK) Functions 13-22
13.4.3.1 KX or KK Read and Write 13-22
13.4.3.2 KX or KK Get Characteristics 13-23
13.4.3.3 KX or KK Enable and Disable 13-24
13.5 Status Codes 13-24
13.6 Communication Driver Prefix Files 13-25
13.6.1 QN Prefix File 13-25
13.6.2 XP and XS Prefix Files 13-26
13.6.3 KX and KK Prefix Files 13-28

13.7 Peripheral Processor Communication Support Routines 13-32
13.7.1 KX_READ_DATA 13-33
13.7.2 KX_WRITE_DATA 13-34
13.7.3 KK_READ_DATA 13-35
13.7.4 KK_WRITE_DATA 13-35

Chapter 14 Guide to Writing a Device Driver

14.1 Device Driver Overview 14-1
14.2 Device Driver Prefix Module 14-3
14.2.1 Priority Assignments 14-3
14.2.2 DRVCF$ Macro 14-4
14.2.3 CTRCF$ Macro 14-5
14.2.4 Sample Driver Prefix Module (DYPFX.MAC) 14-8
14.3 Device Driver Impure-Area Definition Macro (xxISZ$) 14-9
14.4 Device Driver Proper 14-10
14.4.1 Copyright Page 14-11
14.4.2 Module Header 14-11
14.4.3 Functional Description 14-11
14.4.4 Declarations 14-12
14.4.4.1 Local Macro Definition 14-12
14.4.4.2 Externally Defined Symbols 14-12
14.4.4.3 Process Definition 14-12
14.4.4.4 Impure-Area Definition 14-14
14.4.4.5 Pure-Area Definition 14-14
14.4.5 Initialization Process 14-14
14.4.6 Controller Process 14-15
14.4.7 Interrupt Service Routine (ISR) 14-16
14.4.8 Fork Routine 14-17
14.4.9 Reply Subroutine 14-17
14.4.10 Termination Procedure 14-18
14.4.11 Error-Processing Routines 14-18
14.4.11.1 Invalid Requests 14-18
14.4.11.2 Exceptions 14-18
14.4.11.3 Drive or Controller Errors 14-19
14.4.11.4 Resource Famine 14-19

Chapter 15 Device Driver Macros and Subroutines

15.1 Driver Macros 15-1
15.1.1 ADPAR$ (Return PAR Address) 15-3
15.1.2 DRMAP$ (Remap Virtual Address) 15-4
15.1.3 DRPAR$ (Read Contents of PAR or PDR Register) 15-6
15.1.4 DRVDF$ (Define Driver Packet Symbols) 15-7
15.1.5 DSCXW$ (Disable MMU Context Switch) 15-8
15.1.6 DWPAR$ (Write to PAR or PDR Register) 15-10
15.1.7 ENCXW$ (Enable MMU Context Switch) 15-11
15.1.8 IBADR$ (Increment Byte Address and Check for PAR Tick Overflow) 15-13
15.1.9 IWADR$ (Increment Word Address and Check for PAR Tick Overflow) 15-14
15.1.10 MVBYT$ (Move Byte from/to Virtual Addresses) 15-15
15.1.11 MVBYU$ (Move Byte from/to Virtual Addresses from User-Mode) 15-16
15.1.12 MVMAP$ (Move Word from/to Virtual Addresses in Mapped Case Only) . . 15-17
15.1.13 MVVAD$ (Move Address and PAR) 15-18
15.1.14 MVWRD$ (Move Word from/to Virtual Addresses) 15-19
15.1.15 MVWRU$ (Move Word from/to Virtual Addresses from User-Mode) 15-20
15.1.16 SPL$ (Set Priority Level) 15-21
15.1.17 XTAD$ (Compute Bus Extended Address) 15-22
15.2 Driver Subroutines 15-24
15.2.1 $BLXIO (Block Move) 15-25
15.2.2 $DDEXC (Report Exception for Device Driver) 15-26
15.2.3 $DDINI (Device Driver Initialization) 15-27
15.2.4 $DRALR (Allocate Memory) 15-28
15.2.5 $DRDSP (Deallocate Dynamic Memory) 15-29
15.2.6 $DRHIN (Initialize Heap) 15-30
15.2.7 $DRNEW (Allocate Dynamic Memory) 15-31
15.2.8 $DRPLY (Send Device Driver Reply) 15-32
15.2.9 $SV02, $SV03, and $SV05 (Save/Restore Registers) 15-33

Appendix A Directory Structure and File Storage

A.1 Structure of a Random-Access Device A-1
A.1.1 Home Block A-2
A.1.2 Directory A-4
A.1.2.1 Directory Segment Header A-5
A.1.2.2 Directory Entry A-6
A.1.2.3 Extended Directory Entry A-8
A.1.2.4 End-of-Segment Marker A-8
A.2 Directory Use A-9
A.2.1 Sample Directory Segment A-9

A.2.2 Splitting a Directory Segment A-12
A.2.3 File Storage A-16
A.2.4 Method A-16
A.2.5 Size and Number of Files A-18

Appendix B KXT11-CA and KXJ11-CA Peripheral Processors

B.1 KXT11-CA/KXJ11-CA Hardware and Applications B-1
B.1.1 KXT11-CA Hardware Features B-3
B.1.2 KXJ11-CA Hardware Features B-4
B.1.3 Using the KXT11-CA or KXJ11-CA as a Peripheral Processor B-6
B.1.3.1 Peripheral Processor Hardware Configuration B-8
B.1.3.2 Peripheral Processor Application Software Configuration B-8
B.2 Developing KXT11-CA and KXJ11-CA Applications B-9
B.2.1 Partitioning the Application B-9
B.2.2 Designing the Peripheral Processor Application System B-9
B.2.3 Software and Hardware Configuration Guidelines B-10
B.2.3.1 Configuring Memory B-10
B.2.3.2 Memory Configuration Steps B-11
B.2.3.3 Memory Selection Rules B-12
B.2.4 Configuring the KXT11-CA or KXJ11-CA System Environment B-13
B.2.4.1 Selecting Stand-Alone or Peripheral Processor Operation B-13
B.2.4.2 Selecting KXT11-CA or KXJ11-CA Initialization and Self-Test Options . . B-14
B.3 KX/KK Device Driver Communication Protocol B-19
B.3.1 Communication Mechanisms B-19
B.3.2 KX/KK Protocol Definition B-22
B.3.2.1 KX and KK Driver Transactions B-23
B.3.2.2 Message Communication Between the KX and KK Drivers B-25
B.3.2.3 Synchronizing KX and KK Device Driver Operations B-26
B.3.3 Command Register Definition B-27
B.3.3.1 Command Field (KC.COM) B-27
B.3.3.2 Interrupt-When-Data-Available Bit (KC.IDA) B-29
B.3.3.3 Interrupt-When-Data-Requested Bit (KC.IDR) B-30
B.3.3.4 Data Length Field (KC.LEN) B-30
B.3.3.5 End-of-Message Bit (KC.EOM) B-30
B.3.3.6 Vector Number Field (KC.VEC) B-30
B.3.4 Status Register Definition B-30
B.3.4.1 Error Code Field (KS.ERC) B-31
B.3.4.2 Data-Requested Bit (KS.DR) B-31
B.3.4.3 End-of-Message Bit (KS.EOM) B-31
B.3.4.4 Data-Available Bit (KS.DA) B-32
B.3.4.5 Actual Length Field (KS.ALN) B-32

B.3.4.6 Interrupt-Enabled Bit (KS.IEN) B-32
B.3.4.7 Interface-Ready Bit (KS.ON) B-32
B.3.4.8 Cumulative-Error Bit (KS.ERR) B-32
B.3.5 Interface Initialization B-32
B.4 KXT11–CA and KXJ11–CA CSR and Vector Assignments B-33
B.5 System ID Switch Positions, Two-Port RAM CSR and Vector Assignments B-35
B.6 Sample MicroPower/Pascal Configuration File B-37
B.7 Sample Configuration Files for the KXJ11–CA B-40
B.8 Shared Memory on a KXJ B-52
B.8.1 KXJ_ENABLE_SHARED B-53
B.8.2 KXJ_DISABLE_SHARED B-54
B.8.3 Arbiter and KXJ Configuration Files and Applications B-54
B.9 Calculating Checksums for PROMS B-65
B.10 Load Application onto KXT11–CA/KXJ11–CA Procedure B-66
B.10.1 .MIM File B-66
B.10.2 User's Interface B-66
B.10.3 Program Example B-67

Appendix C XL Serial Line Driver

C.1 PDP–11 XL Driver C-1
C.1.1 Functions Provided C-3
C.1.1.1 Read Function C-3
C.1.1.2 Write Function C-3
C.1.1.3 Connect Receive Ring Buffer Function C-4
C.1.1.4 Disconnect Receive Ring Buffer Function C-4
C.1.1.5 Connect Transmit Ring Buffer Function C-4
C.1.1.6 Disconnect Transmit Ring Buffer Function C-4
C.1.1.7 Report Data-Set Status Change Function C-4
C.1.1.8 Set Status Function C-5
C.1.1.9 Get Status Function C-5
C.1.1.10 Device-Independent Function Modifiers C-5
C.1.2 Function-Dependent Request Formats C-5
C.1.2.1 Block-Mode Read or Write Functions C-6
C.1.2.2 Connect Receive or Transmit Ring Buffer Functions C-6
C.1.2.3 Disconnect Receive or Transmit Ring Buffer Functions C-7
C.1.2.4 Set Status Function C-7
C.1.2.5 Get Status Function C-10
C.1.2.6 Report Data-Set Status Change Function C-11
C.1.3 Status Codes C-12
C.1.4 PDP–11 XL Prefix File C-12
C.2 Peripheral Processor XL Driver C-17

C.2.1 Functions Provided C-18
C.2.1.1 Read Function C-19
C.2.1.2 Write Function C-19
C.2.1.3 Connect Receive Ring Buffer Function C-19
C.2.1.4 Disconnect Receive Ring Buffer Function C-19
C.2.1.5 Connect Transmit Ring Buffer Function C-19
C.2.1.6 Disconnect Transmit Ring Buffer Function C-20
C.2.1.7 Set Status Function C-20
C.2.1.8 Get Status Function C-20
C.2.1.9 Report data-set status change function C-20
C.2.1.10 Device-Independent Function Modifiers C-20
C.2.2 Function-Dependent Request Formats C-20
C.2.2.1 Block-Mode Read or Write Functions C-21
C.2.2.2 Connect Receive or Transmit Ring Buffer Functions C-21
C.2.2.3 Disconnect Receive or Transmit Ring Buffer Functions C-22
C.2.2.4 Set Status Function C-22
C.2.2.5 Get Status Function C-25
C.2.2.6 Report Data-Set Status Change Function C-28
C.2.3 Status Codes C-28
C.2.4 KXT11-CA XL Prefix File C-28

Appendix D Sample MACRO-11 Device Driver

Index

Figures

1-1 General I/O Packet Formats 1-12
2-1 ACP Prefix File (ACPPFX.MAC) Excerpt 2-10
3-1 TT Driver Prefix File (TTPFX.MAC) 3-18
4-1 RL01/RL02 Driver Prefix File (DLPFX.MAC) 4-27
4-2 RX02 Driver Prefix File (DYPFX.MAC) 4-28
4-3 MSCP Disk-Class Driver Prefix File (DUPFX.MAC) 4-28
4-4 TU58 Driver Prefix File (DDPFX.MAC) 4-29
4-5 Virtual Memory Driver Prefix File (VMPFX.MAC) 4-30
4-6 Extended Disk Driver Source File (XDDRV.PAS) Excerpt 4-31
5-1 TMSCP Tape Driver Prefix File (MUPFX.MAC) 5-15
6-1 KXT11-CA/KXJ11-CA PIO DMA Sample Program 6-16
6-2 YK Prefix File for PIO DMA Sample Program 6-18
6-3 KXT11-CA/KXJ11-CA External Pulse Counter Sample Program 6-22
6-4 YK Prefix File for External Pulse Counter Sample Program 6-23

6–5 KXT11–CA 32-Bit Counter Sample Program 6-24
6–6 DRV11–J Driver Prefix File (XAPFX.MAC) 6-46
6–7 DRV11 Driver Prefix File (YAPFX.PAS) 6-47
6–8 DRV11–B Driver Prefix File (YBPFX.MAC) Excerpt 6-49
6–9 SBC–11/21 PIO Driver Prefix File (YFPFX.MAC) 6-50
6–10 KXT11–CA/KXJ11–CA PIO Driver Prefix File (YKPFX.MAC) 6-59
7–1 AD Driver Prefix File (ADPFX.MAC) 7-16
8–1 KW Driver Prefix File (KWPFX.MAC) 8-19
9–1 KXT11–CA/KXJ11–CA DMA Sample Program 9-12
9–2 KXT11–CA/KXJ11–CA DTC Driver Prefix File (QDPFX.MAC) 9-26
10–1 Instrument Bus Driver Prefix File (XEPFX.MAC) 10-39
11–1 NSP Prefix File (NSPPFX.MAC) 11-10
12–1 CS Driver Prefix File (CSPFX.MAC) 12-15
13–1 DEQNA Driver Prefix File (QNPFX.MAC) 13-26
13–2 DPV11 Driver Prefix File (XPPFX.MAC) 13-27
13–3 KXT11–CA/KXJ11–CA Synchronous Serial Driver Prefix File (XSPFX.MAC) . . 13-28
13–4 KXT11–CA/KXJ11–CA Two-Port RAM Driver Prefix File (KXPFX.MAC) 13-31
13–5 KXT11–CA/KXJ11–CA Two-Port RAM Driver Prefix File (KKPFX.MAC) 13-32
A–1 Format of Random-Access Device A-2
A–2 Format of Home Block A-3
A–3 Format of Directory Segment A-4
A–4 Format of Directory Entry A-6
A–5 Format of Status Word A-6
A–6 Format of Date Word A-8
A–7 Directory Listing A-9
A–8 Directory Segment A-10
A–9 Storing a New File A-12
A–10 Full Directory Segment A-13
A–11 Directory Before Splitting A-14
A–12 Directory After Splitting A-15
A–13 Directory Links A-16
A–14 Random-Access Device with Two Permanent Files A-17
A–15 Random-Access Device with One Tentative File A-17
A–16 Random-Access Device with Two Tentative Files A-17
A–17 Random-Access Device with Four Permanent Files A-18
B–1 KXT11–CA Hardware Features B-3
B–2 KXJ11–CA Hardware Features B-5
B–3 Adding Peripheral Processors to Traditional LSI–11 Systems B-7
B–4 Peripheral Processor Application Software Configuration B-8
B–5 KXT11–CA Memory Map Configurations B-11
B–6 KX/KK Device Driver Communication Linkage B-20
B–7 TPR Register Layout B-21

C–1 XL Driver Prefix File (XLPFX.MAC) C-16
C–2 KXT11–CA XL Driver Prefix File (XLPFXK.MAC) C-30

Tables

1–1 Request Queue Names, Units, and Unit Numbering 1-9
12–1 Asynchronous DDCMP I/O Paths and Interfaces 12-6
13–1 Communication I/O Paths and Interfaces 13-9
13–2 Two-Port RAM Data Channel Addresses 13-29
13–3 KX Prefix File Defaults 13-29
A–1 Contents of Home Block A-3
B–1 MicroPower/Pascal Usage of KXT11–CA Memory Maps B-12
B–2 Initialization/Self-Test Options for the KXT11–CA B-15
B–3 Initialization/Self-Test Options for the KXJ11–CA B-16

Preface

Intended Audience

This manual describes the MicroPower/Pascal I/O system and the run-time I/O services it provides for user programs. The content of this manual is based on the assumption that you are familiar with either Pascal or MACRO–11. All MicroPower/Pascal microcomputer software development is done with one or both of those development languages. Additional reference information for performing run-time I/O in Pascal is contained in Chapter 9 of the *MicroPower/Pascal Language Guide.*

Structure of This Document

Fifteen chapters and four appendixes make up this manual:

- Chapter 1 presents an overview of the MicroPower/Pascal I/O services. The chapter lists supported devices and protocols, summarizes the I/O system components, mechanisms, and interfaces, and describes the request/reply packet interface to the DIGITAL-supplied device drivers.
- Chapters 2 through 13 describe the DIGITAL-supplied system processes that provide I/O services ("I/O servers"). Chapters 2, 11, and 12 describe the ancillary control process (ACP), the network service process (NSP), and the asynchronous DDCMP protocol driver—that is, the I/O system components that are layered above the device drivers. Chapter 3 describes the asynchronous serial line (terminal) driver, Chapters 4 and 5 the mass-storage device drivers (disk and tape), Chapters 6 through 10 the real-time device drivers (PIO, A/D, DMA, instrument bus), and Chapter 13 the communication device drivers.

  Chapters 2 through 13 describe features and capabilities, application building considerations, user interfaces, completion-status codes, and prefix files for each DIGITAL-supplied I/O server.
- Chapter 14 presents guidelines for writing a MicroPower/Pascal device driver for nonstandard hardware devices—devices not supported by the drivers in the MicroPower/Pascal distribution kit. The chapter describes the necessary components of a device driver and the driver's interface to the application program and refers to sample drivers written in MACRO–11 (DY driver) and Pascal (YA driver).

- Chapter 15 describes macros and subroutines that can be used by device drivers written in MACRO–11.
- Appendix A describes the RT–11-compatible directory structure optionally supported by the MicroPower/Pascal ACP and discusses file storage.
- Appendix B presents information on developing applications for the KXT11–CA or KXJ11–CA peripheral processor.
- Appendix C describes the XL serial line driver, which is included on the MicroPower/Pascal distribution kit for existing applications that require it.
- Appendix D lists the source code for a sample MACRO–11 device driver—the RX02 (DY) driver.

Associated Documents

The following software documentation is required for complete reference purposes:

- MicroPower/Pascal document set
- Standard documentation for your host operating system

You will also need the following hardware reference documents to correctly configure your target (application) hardware, to use the standard device drivers, or to write device drivers that are hardware- and software-compatible with other system components:

- Microcomputer handbooks, including *Microcomputers and Memories* (Order No. EB–20912–20) and *Microcomputer Interfaces Handbook* (Order No. EB–23144–18)
- *SBC–11/21 Single-Board Computer User's Guide* (Order No. EK–SBC01–UG–001), required when developing SBC–11/21 applications
- *KXT11–CA Single-Board Computer User's Guide* (Order No. EK–KXTCA–UG–001), required when developing KXT11–CA applications
- *KXJ11–CA Single-Board Computer User's Guide* (Order No. EK–KXJCA–UG), required when developing KXJ11–CA applications
- *LSI–11 Analog System User's Guide* (Order No. EK–AXV11–46–002), required when developing applications using the ADV11–C, AAV11–C, AXV11–C, or KWV11–C I/O boards
- *IEU11–A/IEQ11–A User's Guide* (Order No. EK–IEUQ1–UG–001), required when developing applications using IEQ11–A instrument bus hardware
- *DPV11 Serial Synchronous Interface Technical Manual* (Order No. EK–DPV11–TM), required when developing applications using DPV11 communication hardware
- *Peripheral Processor Tool Kit–RT Reference Manual* (Order No. AA–AU63C–TC), *Peripheral Processor Software Tool Kit–RSX Reference Manual* (Order No. AA–AU64C–TC), or *Peripheral Processor Tool Kit–MicroVMS Reference Manual* (Order No. AA–HX84A–TE) required when using the KUI utility program to load peripheral processor applications from RT–11, RSX–11, or MicroVMS arbiters

- *VAX/VMS DECprom User's Guide* (Order No. AA-W754A-TK), required when using the VMS DECprom program to calculate and program ROM checksums for KXT11-CA or KXJ11-CA applications
- Additional hardware documentation for microcomputer hardware presently not covered in the microcomputer handbooks

Conventions Used in This Document

1. Pascal-reserved words that must not be abbreviated are shown in uppercase characters in syntax examples. Within those examples, lowercase characters are used for variable parameters (or other syntax elements) that you may choose for your application.
2. In some MACRO-11 syntax diagrams, optional parameters and syntax are shown within brackets ([ ]).
3. Some MACRO-11 syntax examples are shown with long macro invocations continued on a second line—for example, the CRPC$ and DFSPC$ macro calls. However, when writing source code in MACRO-11, you must keep each macro invocation on a single line.
4. This manual uses "MPBUILD" as a generic term for the VMS, RSX, and RT versions of the MicroPower/Pascal automated build procedure. Note that the name of the RT-host version of the procedure is "MPBLD," not "MPBUILD."

Symbols

The numeric values given in this manual for symbols for data structure sizes, offsets, and so forth, are subject to change. Therefore, use symbol names rather than numeric values for components of packets and other system data structures.

Chapter 1

Introduction to MicroPower/Pascal Input/Output

This chapter provides an overview of MicroPower/Pascal input/output (I/O) services. The I/O services include device I/O, task-to-task communication, and Pascal file system operations, including optional RT–11 directory support for disk-class devices. Those services are provided at run time by the MicroPower/Pascal I/O system, consisting of DIGITAL-supplied system processes (called "I/O service processes" or "I/O servers") and the Pascal Object Time System (OTS). The I/O services allow a MicroPower/Pascal program to input data from and output data to devices or tasks that are external to the target processor, using normal Pascal I/O statements or DIGITAL-supplied Pascal support routines.

Note

The *MicroPower/Pascal Run-Time Services Manual* describes run-time services that are provided by the MicroPower/Pascal kernel. Those services include the kernel facilities for interprocess communication (send/receive) and setup of interrupt vectoring (connect-to-interrupt) that are basic to MicroPower/Pascal I/O.

The MicroPower/Pascal I/O service processes support I/O on mass-storage devices, real-time devices, and communication devices.

The supported mass-storage devices and protocols, listed by server, are:

Server	Devices/Protocols
DL driver	RLV11 controller, RL01 disk (16/18-bit addressing) RLV12 controller, RL01/RL02 disks (16/18/22-bit addressing) RLV21 controller, RL01/RL02 disks (16/18-bit addressing)
DY driver	RXV21 controller, RX02 flexible diskettes (single/double density, 18-bit addressing)
DU driver	Mass Storage Control Protocol (MSCP) controllers and disks, including RQDX1, RQDX2, and RQDX3 controllers and RX50, RD51, RD52, RD53, and RC25 disks (22-bit Q-bus environment)
XD driver	Extended (> 65536 blocks) physical disks, partitioned for Pascal I/O
DD driver	TU58 DECtape II connected to DLV or either KXT11-CA or KXJ11-CA serial line interface unit
VM driver	Virtual memory (mapped systems only, requires MMU)
MU driver	TMSCP tapes, including TK50 streaming cartridge tape

The supported real-time devices and protocols, listed by server, are:

Server	Devices/Protocols
XA driver	DRV11-J 64-bit parallel interface (four 16-bit ports)
YA driver	DRV11 16-bit parallel interface
YB driver	DRV11-B DMA interface
YF driver	SBC-11/21 8255 PIO interface
YK driver	KXT11-CA or KXJ11-CA 8-bit parallel ports (16-bit if linked) 4-bit special-purpose I/O port 16-bit counter/timers
AD driver	ADV11-C and AXV11-C A/D converter boards
KW driver	KWV11-C programmable real-time clock
QD driver	KXT11-CA or KXJ11-CA 2-channel direct transfer controller (DTC)
XE driver	IEQ11-A instrument bus interface

The supported communication devices and protocols, listed by server, are:

Server	Devices/Protocols
TT driver	Asynchronous serial (terminal) lines, including DLV11-type (DLV11, DLV11-E, DLV11-F, DLV11-J), DLART-type (KXT11-CA or KXJ11-CA console, SBC-11/21, CMR21, MXV11-A, MXV11-B), DZV11, DHV11, KXT11-CA or KXJ11-CA multiprotocol chip
CS driver	DDCMP over asynchronous serial lines (usable as base for DECnet)
QN driver	DEQNA Ethernet interface, Ethernet data link protocol (usable as base for DECnet)
XP driver	DPV11 synchronous serial line interface, bit-synchronous mode (usable as base for bit-oriented protocol, such as HDLC or LAPB)
XS driver	KXT11-CA or KXJ11-CA synchronous serial line interface (usable as base for bit-oriented protocol)
KK driver	KXT11-CA or KXJ11-CA two-port RAM, peripheral processor side of two-port RAM protocol
KX driver	KXT11-CA or KXJ11-CA two-port RAM, arbiter side of two-port RAM protocol

The servers that are not specific to a single device or protocol—the ACP and the NSP—do not appear above. However, the ACP supports all the OPENable devices and protocols among those listed, and the NSP supports all listed communication devices and protocols (including, indirectly, TT).

1.1 I/O System Architecture

The MicroPower/Pascal I/O system has the following components:

- Pascal/file system OTS
- Ancillary control process (ACP)
- Network service process (NSP)
- Protocol/device drivers

Note

The required participants in a standard (driver-based) I/O transfer are a calling user process, a device driver, and an appropriately set-up hardware device. For task-to-task communication, a partner task on a remote processor is also required. The other components—OTS, ACP, NSP, protocol driver—are layered on the device driver (and each other) and function as intermediaries in an I/O transfer.

The Pascal OTS is composed of the Pascal kernel and I/O system interface routines. The I/O system interface routines reside in a separate file and are called the file system OTS. The OTS routines are built into a user process on an as-needed basis—automatically if you build with MPBUILD. (If building without MPBUILD, you must include the appropriate OTS libraries on the MERGE utility command line.)

In contrast to the OTS routines, which can be viewed as part of the user process that requests an I/O service, the ACP, the NSP, and the protocol/device drivers are system processes, termed I/O service processes or I/O servers. You build the required system processes into your application by editing and assembling a system-process prefix module. That module includes a global symbol reference that causes the appropriate system process to be merged into the application.

The user process, the ACP, the NSP, and the drivers communicate with each other via the kernel mechanisms for interprocess communication—the high-level (send/receive) or low-level (signal-queue/wait-queue) queue semaphore kernel primitives.

The ACP supports file-opening operations for I/O devices and protocols plus standard Pascal I/O on disk devices. (Optionally, the ACP supports RT-11 file structure on disk devices or opening of task-to-task links.) The ACP is called when an application program opens a file. The open operation associates a file variable with an I/O server (a driver, the ACP for disk operations or the NSP for task-to-task operations), making it possible to perform normal device-independent Pascal I/O via the server. Subsequent I/O requests go directly to the server.

The NSP supports task-to-task communication between a MicroPower/Pascal application and an application on a remote processor. The NSP is called (by the ACP) when an application program opens a logical link, over a physical communication link, with a remote task. The open operation associates a file variable with an NSP logical-link server, making it possible to perform device-independent Pascal task-to-task I/O. Subsequent I/O requests go directly to the server.

The protocol/device drivers support I/O on a protocol or a hardware device. (The current version of MicroPower/Pascal has only one protocol driver—for asynchronous DDCMP.) The driver is called by the user process (that is, the OTS or alternative routines), the ACP (for OPEN or disk I/O), or the NSP (task-to-task I/O) as necessary to complete a user-requested I/O operation.

The device drivers normally communicate with and control the hardware by manipulating device registers or other I/O page locations. In addition, the drivers establish hardware-interrupt vectoring via the connect-to-interrupt kernel primitive. When a hardware device issues an interrupt to signal completion of a transfer or to request further transfer-related processing from the driver, control is passed to an interrupt service routine (ISR) in the driver. The ISR performs critical processing in kernel mapping context at a high priority, then issues a FORK$ call for less-critical processing or kernel-primitive invocation (possibly signaling a driver semaphore), then exits, allowing interrupted or lower-priority processing to continue.

The MicroPower/Pascal I/O system provides three basic user interfaces to I/O:

- Pascal file system I/O (normal Pascal I/O statements)
- Pascal support routines (independent of file system)
- Request/reply packet I/O (send/receive)

The request/reply packet interface uses the kernel send/receive primitives to issue requests directly to the request queue semaphore of the ACP, the NSP, or a driver. The request/reply packet interface is the central mechanism for MicroPower/Pascal I/O and provides the basis for the higher-level file system and support routine interfaces.

The rest of this section summarizes the possible I/O request paths (user process —> device driver) through the I/O system.

Note

The following abbreviations are used in this section:

U = User process

FSOTS = File system OTS routines

ACP = Ancillary control process

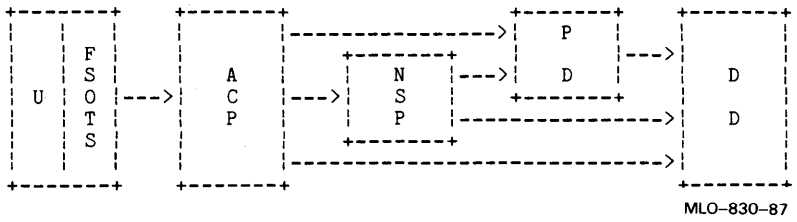
NSP = Network service process

PD = Protocol driver (CS)

DD = Device driver

Each arrow represents a kernel send or signal-queue operation.

The possible I/O request paths for Pascal OPEN operations are shown below:

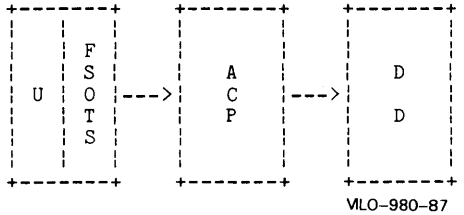


MLO-830-87

The paths shown above correspond to the following OPEN operations (x denotes a participant in the operation):

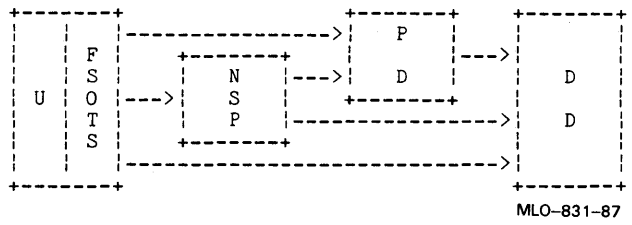
U	FSOTS	ACP	NSP	PD	DD	Operation
x	x	x	x	x	x	OPEN of NSP/CS/TT link
x	x	x	x		x	OPEN of NSP/DD link
x	x	x		x	x	OPEN of CS/TT file
x	x	x			x	OPEN of openable DD file

The I/O request path for a Pascal I/O operation on an opened disk file is shown below:



MLO-980-87

The possible I/O request paths for Pascal I/O operations on opened nondisk files or logical links are shown below:



The paths shown above correspond to the following operations (x denotes a participant in the operation):

U	FSOTS	ACP	NSP	PD	DD	Operation
x	x		x	x	x	Pascal I/O on NSP/CS/TT link
x	x		x		x	Pascal I/O on NSP/DD link
x	x			x	x	Pascal I/O on CS/TT file
x	x				x	Pascal I/O on nondisk DD file

To perform nonfile I/O from a MACRO-11 program—or a Pascal program from which you wish to exclude the FSOTS, the ACP, the NSP, and any Pascal support routines—you must issue send requests to a driver request queue semaphore. The following operations can be performed:

U	FSOTS	ACP	NSP	PD	DD	Operation
x				x	x	CS/TT function
x					x	DD function

To perform file I/O from a MACRO-11 program—or a Pascal program from which you wish to exclude the FSOTS—you must issue a send request to the ACP to open the file or logical link. (If you wish to exclude the ACP for a logical link open, it is still considered a file system operation, so you must issue an ACP-formatted send request to the NSP.) Subsequent send requests—for read, write, and so forth—must be issued to the ACP or driver queue semaphore identified in the reply to the open request.

The following operations can be performed (x in parentheses denotes an optional participant in the operation):

U	FSOTS	ACP	NSP	PD	DD	Operation
x		(x)	x	x	x	OPEN of NSP/CS/TT link
x		(x)	x		x	OPEN of NSP/DD link
x		x		x	x	OPEN of CS/TT file
x		x			x	OPEN of DD file or disk file I/O
x			x	x	x	I/O on opened NSP/CS/TT link
x			x		x	I/O on opened NSP/DD link
x				x	x	I/O on opened CS/TT file
x					x	I/O on opened nondisk DD file

Note

With regard to the ACP and NSP entries above, note that the current version of this manual does not provide detailed descriptions of the ACP and NSP send/receive interfaces.

1.2 Performing I/O

For most MicroPower/Pascal applications, you perform I/O in one of two ways. You can invoke Pascal I/O statements that open files for data and then input or output the data, in accordance with the rules for Pascal I/O. The Pascal I/O procedures—OPEN, GET, WRITE, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide.*

For drivers that do not permit file system access—for example, QD, or XE—or for which file access is of limited usefulness—for example, MU, YK, or KW—you perform I/O by calling DIGITAL-supplied support routines that are independent of the file system. Those routines provide high-level nonfile access to an I/O resource. The routines typically issue Pascal SEND requests to the request queue semaphore of a device driver. The support routines are described in detail in Sections 5.3 (MU), 6.4 (YF/YK), 7.4 (AD), 8.3 (KW), 9.3 (QD), 10.4 (XE), and 13.7 (KX/KK).

In addition to invoking the Pascal I/O statements or support routines, you must:

1. [For each device driver:] Edit the DEVICES macro in the system configuration file to reflect the device interrupt vector addresses
2. Edit the prefix file for each required system process, as described in the prefix file sections of Chapters 2 through 13
3. Build into your application the required I/O system components:
   - Driver process(es)
   - [For file OPEN:] ACP
   - [For logical link OPEN:] NSP

- Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—or nonfile-oriented support routines, plus any other I/O routines you opt to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, the prefix file sections of Chapters 2 through 13, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

The I/O system file system and support routine interfaces conceal from the Pascal user the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to I/O server queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores.

Note

It is possible to bypass the file system, the ACP, and any available support routines in order to access a device driver directly. This can be accomplished via send/receive operations to a driver's request queue semaphore.

It is also possible, given detailed knowledge of the ACP and NSP request/reply packet interfaces, either to bypass the file system OTS in order to access the ACP directly, or to bypass the file system OTS and the ACP to access the NSP directly. However, the current version of this manual does not provide detailed descriptions of the ACP and NSP send/receive interfaces.

1.3 Request/Reply Packet Interface

I/O servers are system processes that accept requests for I/O operations from user or system processes. DIGITAL-supplied I/O servers include device drivers and layered processes, such as the protocol (DDCMP) driver, the network service process (NSP), and the ancillary control process (ACP). The mechanism for interprocess communication is the kernel send/receive (or lower-level signal/wait) queue semaphore facility. I/O requests for a device or service are passed to the server in the form of a request message (queue packet). Each server maintains a request queue semaphore, through which I/O requests are passed. The request packet supplies all the information the server needs to perform the desired operation, including the function code, type of reply desired, and where applicable, the unit number, device address, and data-buffer location. After receiving a request, a device-level server (device driver) will perform all process-level, interrupt-level, and fork-level processing for the requesting process; a layered server (ACP, NSP, or protocol driver) will perform processing and give requests to other layers as necessary to complete the operation.

When the I/O operation has been completed, if a full reply was requested, the server signals the requesting process and returns a reply message packet (often a modified version of the request packet). The reply message packet indicates completion status and other information, such as number of bytes successfully transferred, as applicable.

This section describes the general features of the send/receive I/O packet interface as those features pertain to DIGITAL-supplied drivers (see note). The device- or function-dependent aspects of the I/O packet interface are covered in the individual driver descriptions in Chapters 3 through 10 and Chapters 12 and 13.

Note

The device driver request and reply packets are described later in this section and throughout the driver chapters. The symbols used to describe the packets and the information they contain are MACRO-11 symbols defined by the kernel macro DRVDF$ from the COMU/COMM kernel macro libraries. The Pascal equivalents of those symbols are defined in IOPKTS.PAS, an include file that is recommended for use with Pascal I/O requests.

The ACP and NSP packet-level interfaces are not documented in detail in the current version of this manual.

1.3.1 Request Queue Names

The driver request queue semaphores have standardized, 4-character names that identify the driver associated with the semaphore and the controller serviced by the driver. The names are of the form $ddc:

Designator	Meaning
dd	A driver identifier (for example, DY for RX02 or TT for terminal line)
c	A controller identifier (for example, A, B, C, ...—as specified in a driver prefix file—or simply A where multiple controllers do not apply)

Thus, $DYA and $DYB would name the request queues for the first and second RX02 controllers configured on a system, and $TTA would name the queue for any asynchronous serial line interface.

The request queue name must be specified in uppercase letters. Also, since device drivers specify 6-character names, including two space characters, you should space-fill the last two character positions in the request queue name when creating the request queue.

Table 1-1 lists standard request queue names, supported hardware units, and unit numbering for standard device drivers.

Table 1-1: Request Queue Names, Units, and Unit Numbering

Driver	Request Queue Name	Number of Units	Numbering
Asynchronous serial	$TTA	1-n (1-4 for DZV11, 1-8 for DHV11, 1 for most others)	0 through (n-1) in prefix file order, crossing controller boundaries
RL01/2	$DLc	1-4 (any combination of RL01s and RL02s)	In prefix file
RX02	$DYc	1-2	0 for left drive and 1 for right in dual-drive
MSCP	$DUc	1-n	In prefix file

Table 1–1 (Cont.): Request Queue Names, Units, and Unit Numbering

Driver	Request Queue Name	Number of Units	Numbering
Extended disk	$XDc	1–n (partitions), as determined by disk size and user-defined partition size	0 through (n–1)
TU58	$DDc	1–2	0 for left drive and 1 for right in dual-drive
Virtual memory	$VMc	1	0
TMSCP	$MUc	1	0
DRV11–J	$XAc	[For read/write:] 1–4 [For Enable/Disable:] 1–12	0 through 3 for ports A through D 4 through 15 for port A lines 0 through 11
DRV11	$YAA	1	0
DRV11–B	$YBc	1	0
SBC–11/21 PIO	$YFA	1–2	0 and 1 for ports A and B
KXT11–CA or KXJ11–CA PIO	$YKA	1–6	0 through 5 for ports A through C and timers 1 through 3
A/D converter	$ADc	1	0
Real-time clock	$KWc	1	0 (normally)
KXT11–CA or KXJ11–CA DMA	$QDc	1–2	0 and 1 for channels A and B
Instrument bus	$XEc	1 (per controller)	Sequentially upward from 0 in prefix file order, crossing controller and board boundaries
DDCMP	$CSA	1–n	0 through (n–1) in prefix file order, independently of TT unit numbers
DEQNA	$QNc	1–4 (portals)	In prefix file
DPV11	$XPc	1	0
KXT11–CA or KXJ11–CA synchronous serial	$XSc	1	0
KXT11–CA two-port RAM	$KXc	1–2	0 and 1 in prefix file order
KXT11–CA two-port RAM	$KKA	1-2	0 for channel 0 and 1 for channel 1

1.3.2 I/O Request and Reply Packets

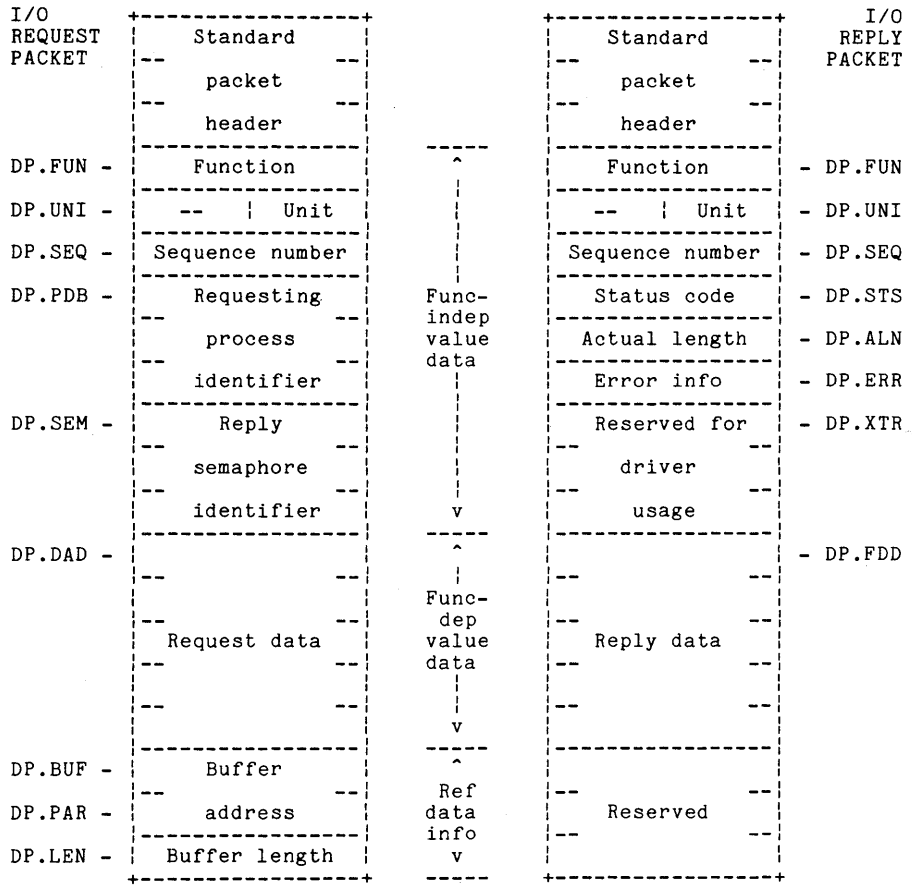
Figure 1–1 shows the general form of an I/O request packet as received by the driver and an I/O reply packet as received by the caller. The diagram includes the standard 6-byte header that prefixes all packets and that is transparent to users of the send/receive-level mechanisms. (That header is provided by the SEND$ primitive, based on kernel information and user-provided macro arguments, when it builds the packet; it should not be included in the send or reply buffers that are specified in the send/receive calls.)

Note that the request data consists of an 18-byte portion that is function-independent—fields DP.FUN to DP.SEM—and a 16-byte portion that varies in content, depending on the kind of function requested—fields DP.DAD to DP.LEN.

Note

The field names shown do not represent offsets into the send or reply buffers; rather, they correspond to offset symbols used by the drivers to reference packets; for example, DP.FUN is a 6-byte offset from the packet header.

Figure 1–1: General I/O Packet Formats



MLO-832-87

The request packet fields shown in Figure 1–1 have the following significance:

Field	Significance
DP.FUN	The 6-bit function code and the function-modifier bits that together specify the operation to be performed. The function word is divided into three subfields, as follows: • Function code value in bits 0 to 5: IF$RDP = Read Physical IF$RDL = Read Logical IF$WTP = Write Physical IF$WTL = Write Logical IF$SET = Set Characteristics IF$GET = Get Characteristics Other codes denote device-specific functions—for example, IF$SMD, as used in the TT driver. See individual device-driver descriptions for device-specific functions. The following function codes are reserved (see Sections 2.4.5 through 2.4.7): IF$LOK = Lookup (open) IF$ENT = Enter (open) IF$REN = Rename IF$DEL = Delete IF$CLS = Close IF$PRG = Purge IF$PRO = Protect IF$UNP = Unprotect • Device-dependent function-modifier bits 6 to 12 (their meaning is described separately for each driver). • Device-independent function-modifier bit settings, for bits 13 to 15: FM$BSM (bit 13) Set = Reply semaphore (DP.SEM) is a binary or a counting semaphore FM$DCK (bit 14) Set = Data check FM$INH (bit 15) Set = Inhibit retry of soft device errors
DP.UNI	The unit number of the desired device, where applicable. (The high-order byte of DP.UNI is reserved.) See Table 1–1 for unit-numbering information.
DP.SEQ	An optional, user-defined value, for example, a sequence number for identifying a given request. This field is provided for the user's purposes only; it is not used by the driver but is returned in the reply packet.
DP.PDB	The Pascal STRUCTURE_ID-type variable that identifies the requesting process (first three words of the process descriptor block; see Section 3.1.6 of the *MicroPower/Pascal Run-Time Services Manual*). This field is used for the QD driver Allocate Channel (IF$ALL) function; see Chapter 9.

Field	Significance
DP.SEM	The Pascal STRUCTURE_ID-type variable that identifies the user's completion-reply semaphore (first three words of the structure descriptor block; see Section 3.1.5 of the *MicroPower/Pascal Run-Time Services Manual*). If modifier bit FM$BSM of word DP.FUN is not set, implying a full reply, this field must identify a queue semaphore through which a reply packet is to be sent. If modifier bit FM$BSM is set, this field can identify either a binary or a counting semaphore, which is signaled on request completion (whether successful or not). If the first word of this field is zeroed, the driver takes no completion-reply action.
DP.DAD	Interpreted according to the type of device handled and the function requested and is unused in some cases. For logical I/O on a disk, the first two words are interpreted as a double-word logical block number, with the least-significant part in the first word. Other drivers either ignore this field or interpret it differently (see the TT driver Get Characteristics request packet in Chapter 3, for example).
DP.BUF	The virtual address of the start of the user's data buffer. This word is filled in automatically by SEND or SEND$, based on the reference-buffer parameter you supply in the call.
DP.PAR	The page address register value that maps the user's data buffer. This value is supplied and filled in automatically by SEND or SEND$ and is meaningful only in a mapped environment.
DP.LEN	The amount of data to be transferred, in bytes. This word is filled in automatically by SEND or SEND$, based on the reference-length parameter you supply in the call.

Note

If not used for reference data information, fields DP.BUF through DP.LEN can be used for additional value data.

Note that all drivers notify the requesting process of a request completion, if a reply semaphore is specified in the request (DP.SEM is nonzero), by either a full reply or a done signal, as determined by function-modifier bit FM$BSM of the function word (DP.FUN). If bit FM$BSM is not set, a full reply (also shown above) is sent via the queue semaphore identified in DP.SEM.

If bit FM$BSM is set, the binary or the counting semaphore identified in DP.SEM is signaled on request completion. In this case, the requesting process cannot determine whether the operation completed successfully. If the requesting process does not desire any notification of completion, the first word of DP.SEM must contain 0, in which case the setting of DP.FUN bit 13 is not significant.

The function-dependent portion of a request is described in detail for each driver in the individual driver descriptions.

The reply message is a modified form of the request message, with the DP.FUN and DP.SEQ fields unchanged and the following fields filled in as appropriate:

- DP.STS (DP.PDB), in which completion-status information has been inserted
- Possibly DP.ALN and DP.ERR (DP.PDB+2 and DP.PDB+4), in which the actual length of a transfer and error information may have been placed
- Possibly DP.XTR (DP.SEM)
- Possibly some portion of the function-dependent value data field, DP.FDD

The meanings of the modified fields in the reply message shown in Figure 1–1 are as follows:

Field	Significance
DP.STS	Code for completion status, indicating type of error; the exception codes returned are listed in Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual* and in the individual driver chapters; ES$NOR (0) indicates success
DP.ALN	The length of the data actually transferred, in bytes, for transfer functions
DP.ERR	Device-dependent hardware- or software-error information if DP.STS is nonzero
DP.FDD	To be interpreted according to the type of device handled and the function requested—unused in most cases

Individual driver descriptions in later chapters provide more specific information about the status, length, and error word values and function-dependent information in the DP.FDD field.

Chapter 2
Ancillary Control Process

This chapter describes the MicroPower/Pascal ancillary control process (ACP), which in cooperation with the network service process (NSP), the standard I/O drivers, and the Pascal file system OTS (or equivalent user routines), provides the capability for device-independent file I/O. (The ACP, the NSP, and the drivers are collectively referred to as "I/O service processes" or "I/O servers.") Also, the ACP optionally provides RT-11 directory services, which allow you to set up RT-11-compatible file directory structures on disk devices.

2.1 ACP Features and Capabilities

The ACP supports file-opening operations for all MicroPower/Pascal-supported I/O devices and protocols plus normal Pascal I/O on disk devices. It is called from a MicroPower/Pascal application program in order to associate a file variable with an I/O service process, making it possible to perform device-independent I/O via normal Pascal I/O statements. Requests for file-opening or disk-transfer operations are passed by the file system OTS to the ACP.

The functions of the ACP include:

- Parsing user device/file specifications
- Determining device characteristics
- Supporting RT-11 file structure on disk-class devices (optionally enabled in prefix file)
- Checking file limits when accessing disk-class devices
- Supporting parsing for task-to-task links in cooperation with the NSP—a DECnet Session Control layer function (optionally enabled in prefix file)

2.2 Accessing the ACP for File I/O

For most MicroPower/Pascal applications, you access the ACP implicitly by opening a file with the Pascal OPEN statement. If the file in question is a named file on a directory-structured disk, other Pascal I/O statements you issue implicitly access the ACP—BREAK, DELETE_FILE, and so forth. See Section 2.3 for more information on the Pascal file system interface to the ACP; see Chapter 9 of the *MicroPower/Pascal Language Guide* for descriptions of OPEN and the other Pascal I/O statements.

In addition to issuing the OPEN and subsequent Pascal I/O procedure calls, you must:

1. Edit the ACP prefix file to indicate:
   - ACP initialization and request-handling process priorities
   - Directory operation priority
   - Whether RT–11 directory support is required
   - Whether network open support is required
   - The ACP dynamic pool size
2. Build into your application the following I/O system components:
   - ACP process
   - I/O service processes (device drivers and NSP, as appropriate) to be accessed via file system (OPEN)
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any support routines you opt to include (see kit files FSPAS.PAS, INTDIR.PAS, GETSET.PAS, and GSINC.PAS)

For more information on setting up your application software for file system I/O, see Section 2.6, the NSP and driver chapters of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

Note

It is possible to bypass both the file system and the ACP in order to access a device driver directly. This can be accomplished via send/receive operations to a driver's request queue semaphore or, in some cases, via DIGITAL-supplied support routines that talk to a particular driver. Such access is referred to throughout this manual as "nonfile access."

As noted in other chapters, it is also possible, given detailed knowledge of the ACP and NSP request/reply packet interfaces, to either bypass the file system OTS in order to access the ACP directly or bypass the file system OTS and the ACP to access the NSP directly. Such access is referred to as "low-level file system access." However, the current version of this manual does not provide detailed descriptions of the ACP and NSP send/receive interfaces.

The following sections describe the Pascal file system interface to the ACP, the lower-level request/reply packet interface (in general terms—see the preceding note), the status codes that can be returned to users of either interface, and the ACP prefix file. An application note on device-name parsing concludes the chapter.

2.3 Pascal File System Interface

The following Pascal I/O statement implicitly accesses ACP services:

OPEN

When you open a non-directory-structured file—that is, a file that does not have a directory, such as a terminal line, a communications port, a network link, or an A/D converter—the file system OTS sends an open request to the ACP, and the ACP sends the request to the associated I/O server (device driver or NSP) for any device-dependent open processing. When the device/server completes open processing, it replies to the ACP. Provided no error occurred, the ACP returns the unit number and the structure descriptor block (SDB) of the I/O server to the file system OTS. All subsequent operations to that file are sent by the file system OTS directly to the I/O service process, with no further ACP involvement.

However, when you open a disk file, whether directory-structured or not, the ACP associates a channel with your file variable and returns the channel number and the ACP's SDB to the file system OTS. All subsequent operations to that disk file are processed by the ACP. This allows the ACP to perform file-limit checks for disk files. If you build RT-11 directory support into your application—by specifying RTSUP = 1 in the ACP prefix file—RT-11 directory operations can be performed.

The current OTS and ACP interaction does not allow for Pascal I/O with disks having greater than 65,536 blocks. I/O transfer computations are performed with 16 bits, with no allowances made for media having block counts that exceed 16 bits. If multiblock GET transfers are being performed to a disk opened as 'DUA0:' or 'XDA1:', for example, the ACP may not detect when the 16-bit block count overflows and wraps around (beginning again at zero). Thus, EOF is never returned, and the operation loops.

The following Pascal I/O statements implicitly access the ACP for disk-class devices only:

DELETE_FILE	BREAK
INIT_DIRECTORY	CLOSE
PROTECT_FILE	EMPTY_BUFFER
RENAME_FILE	GET, READ
SQUEEZE_DIRECTORY	PUT, WRITE
UNPROTECT_FILE	PURGE

GET, READ, PUT, and WRITE statements may or may not trigger ACP requests, depending on the current state of the OTS buffers. The appropriate request packets are sent to the ACP only when necessary to complete a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the ACP. Other Pascal statements unconditionally cause the OTS to issue send requests; examples are BREAK, which generates a Write Logical (IF$WTL), and CLOSE, which generates a Close (IF$CLS) request (normally preceded by a Write Logical, unless BREAK immediately precedes CLOSE).

Pascal Get Characteristics functions that report the characteristics of disks are provided in the file GETSET.PAS in the MicroPower/Pascal distribution kit. Those functions issue Get Characteristics (IF$GET) request packets to the driver.

2.4 Request/Reply Packet Interface

The packet-level functions provided by the ACP are listed below by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical
IF$RDL (1)	Read Logical
IF$WTP (3)	Write Physical
IF$WTL (4)	Write Logical
IF$SET (6)	Set Characteristics
IF$GET (7)	Get Characteristics
IF$LOK (16)	Lookup
IF$ENT (17)	Enter
IF$REN (18)	Rename
IF$DEL (19)	Delete
IF$CLS (20)	Close
IF$PRG (21)	Purge
IF$PRO (22)	Protect
IF$UNP (23)	Unprotect

Many of the functions are not processed directly by the ACP but rather are passed to the I/O service process connected to the channel.

Note

When a disk-class device is opened, a channel is allocated in the ACP, and subsequent requests for that device come to the ACP. When a nondisk device is opened, the ACP is called only for the open. Subsequent requests for that device go directly to the device driver or service process with no further ACP involvement.

The ACP consists of an initialization process, which lowers its priority to become the main request server. The main request server handles all I/O requests for open disk files and passes all open or RT-11 directory requests to the RT-11 directory process. The RT-11 process performs device-specification parsing, determination of device characteristics, and all RT-11 directory operations.

2.4.1 Physical Read and Write Functions

Physical read and write requests are valid only on an open channel. The request is sent to the device driver with no limit or boundary checks.

2.4.2 Logical Read and Write Functions

Logical read/write requests are checked for validity (within bounds of the file for a file-structured device, channel open verification) and the request is passed on to the appropriate driver. The driver will reply directly to the user upon completion of the request. On write operations, the ACP will update the current high mark in the file if the file was entered. The request packet for read/write requests is the same as for device drivers, except that the unit (DP.UNI) field contains a channel number.

Each read or write request specifies the beginning block number for the operation.

If the read or write operation causes the limits of a file on a file-structured device to be exceeded, the ACP truncates the transfer length to the maximum available in the file. If no data is available, status ES$EOF is returned.

2.4.3 Set Characteristics Function

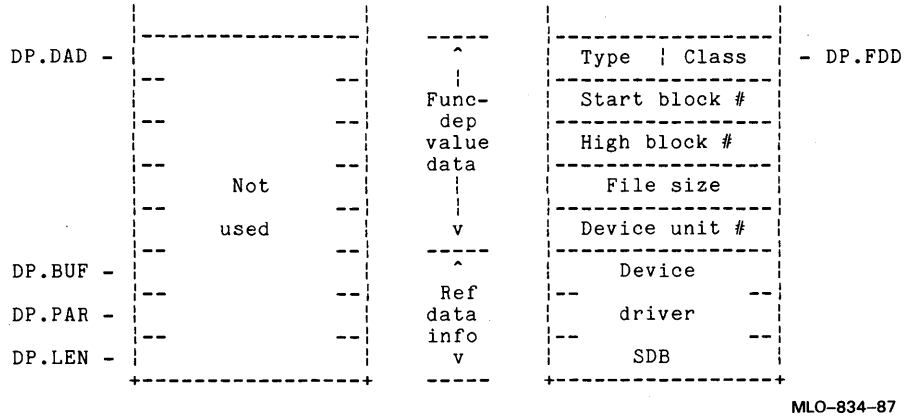
Set Characteristics requests are valid only on an open channel. The request is sent to the device driver with no checks.

2.4.4 Get Characteristics Function

The Get Characteristics function returns a block of device-dependent information about a specified channel in the function-dependent portion of the reply message. The information consists of the codes for device class, type, starting block number of the file, highest block in use, the file size, the device driver unit number, and the device driver structure descriptor block.

If no channel is specified (packet offset DP.UNI) or if the specified channel is not open, the ACP returns only a class and type code indicating that the response has come from the ACP.

The function-independent portions of the Get Characteristics request and reply packets are the same as for device drivers, except that, as noted above, the unit (DP.UNI) field contains a channel number. The function-dependent portions of the packets are as follows:



In the reply information:

- Class is DC$SSV, for system service class.
- Type is SS$DFL for file (directory) structured access, SS$DSK for nonfile (nondirectory) structured access, or SS$ACP if no channel or an invalid channel was specified.

If the type is SS$DFL or SS$DSK the following information is returned:

- The starting block number of the file on the disk
- The highest logical block number used within the file (normally meaningful only when creating a file—that is, HISTORY := NEW)
- The size of the file in blocks
- The unit number of the device on which the file resides
- The device driver structure descriptor for the device on which the file resides

Note

The MACRO–11 field names do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 I/O servers to reference packets. For example, DP.FDD is a 24-byte (decimal) offset from the packet header. The symbols are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

2.4.5 Lookup and Enter Functions

Lookup and Enter are the OPEN functions. For directory-structured I/O, Lookup is used to find an existing file, and Enter is used to create a new file. For network I/O, Lookup designates the active task, and Enter establishes a passive task. For all other I/O, Lookup and Enter are equivalent.

Lookup and Enter parse the user's file specification. If the device specification is a ring buffer, the SDB of the ring buffer is returned to the file system OTS. The file system then operates directly on the ring buffer. Otherwise, the ACP sends a Get Characteristics request (IF$GET) to the I/O server request semaphore in order to determine the device characteristics.

If the Get Characteristics succeeds, the ACP passes the Lookup or Enter request to the I/O server to allow it to perform any device-specific open processing. I/O servers must reserve Lookup and Enter function codes even if they do not implement those functions. I/O servers may ignore the requests if they have no device-specific processing to perform for them. I/O servers that ignore the requests should return them with ES$NOR (normal completion) or ES$IFN (invalid function) status. The ACP interprets ES$IFN as indicating that no special processing was required and continues processing the request as if ES$NOR were returned. (This allows compatibility with Version 1 MicroPower/Pascal drivers.) Alternatively, an I/O server may support Lookup or Enter, performing appropriate device-specific open processing. However, if an I/O server does not wish to be accessed by the ACP, it should return ES$UFN (unsupported function) or any other error code (other than ES$IFN), informing the ACP that an error occurred during open processing.

If no error occurs in device-dependent open processing, the ACP returns the following information to the file system OTS:

- For nondisk devices, the unit number and the device-driver SDB
- For disk devices, a channel number and the ACP SDB

The file system OTS sends all subsequent requests for the specified device to the I/O server indicated in the ACP reply. Thus, disk requests are sent to the ACP, while nondisk requests are sent to a device driver with no further ACP involvement.

2.4.6 Rename, Delete, Protect, and Unprotect Functions

Rename, Delete, Protect, and Unprotect are valid only on permanent files. A permanent file is one with the PERM bit set in the directory entry; see Appendix A for more information on RT–11 directory structure.

The ACP searches the directory for the specified file and, if the file is found, changes the directory entry. (Note that Delete changes the status of the file from permanent to empty.)

No checks are made to determine if the file is currently open for another user; the ACP does not perform any contention checks on files.

2.4.7 Close and Purge Functions

Close and Purge are valid only for a channel that has been defined by a previous Lookup or Enter request to the ACP.

Close makes a tentative file permanent if the file was Entered. If the file was opened with a Lookup, Close is functionally equivalent to Purge (deallocates channel).

Purge makes a tentative file empty if the file was Entered. If the file was opened with a Lookup, Purge deallocates the channel in the ACP.

Any further requests on the channel after Close or Purge are invalid, since the channel is no longer defined.

The file system OTS passes Close/Purge requests on to the I/O server when the Pascal CLOSE/PURGE procedures are executed. I/O servers must reserve Close and Purge function codes even if they do not wish to implement the functions. I/O servers may completely ignore these requests if they have no device-specific processing to perform for any of them. I/O servers that ignore the requests should return them with an ES$NOR (normal completion) or ES$IFN (invalid function) status. ES$IFN indicates to the file system OTS that no special processing was required. (This allows compatibility with Version 1 MicroPower/Pascal drivers.) Alternatively, an I/O server can support Close or Purge, performing appropriate device-specific Close/Purge processing.

2.5 Status Codes

The ACP returns the exception codes shown below in the status-code field of the reply message. If you perform I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will report the corresponding exception (unless the operation was an OPEN, DELETE_FILE, RENAME_FILE, PROTECT_FILE, UNPROTECT_FILE, INIT_DIRECTORY, or SQUEEZE_DIRECTORY for which a STATUS return was specified). The error codes shown are those generated by the ACP directly—not those generated by other I/O system components involved in file I/O.

If no error is detected during the I/O operation, the ACP returns a value of ES$NOR (0) in the status-code field.

The following codes are returned by all configurations of the ACP:

Code	Type	Description
ES$ABT	HARD_IO	I/O request canceled or aborted
ES$NXU	HARD_IO	Nonexistent unit or channel
ES$DVF	SOFT_IO	Attempt to signal device driver failed
ES$EOF	SOFT_IO	End of file encountered
ES$IDS	SOFT_IO	Illegal device specification
ES$IFN	SOFT_IO	Illegal function
ES$IFS	SOFT_IO	Illegal file specification
ES$IRS	SOFT_IO	Illegal rename specification
ES$NFS	SOFT_IO	Device not file structured
ES$NRF	SOFT_IO	No reference data present
ES$WEF	SOFT_IO	Attempted write past EOF
ES$NMC	RESOURCE	Insufficient space for operation in ACP pool

The following codes are returned only if RT-11 directory support is selected (RTSUP = 1) in the prefix file:

Code	Type	Description
ES$DCF	SOFT_IO	Device full
ES$DIO	SOFT_IO	Directory I/O error
ES$DRF	SOFT_IO	Directory full
ES$FNF	SOFT_IO	File not found
ES$IDR	SOFT_IO	Invalid directory format
ES$PRO	SOFT_IO	File protection error

The following codes are returned only if NSP support is selected (NSPSUP = 1) in the prefix file:

Code	Type	Description
ES$INS	SOFT_IO	Invalid network specification
ES$NNS	RESOURCE	No network service process installed

Exception codes are defined in the EXC.PAS include file for Pascal users and by the EXMSK$ macro in the COMU and COMM macro libraries for MACRO–11 users.

2.6 ACP Prefix File

Figure 2–1 shows the user-modifiable portion of the ACP prefix module. The following paragraphs describe the macro calls and symbol definitions that can be edited to fit your application.

The ACP prefix file allows you to enable or disable RT–11 file support, enable or disable network OPEN support, and tune the size of the ACP pool area.

RT–11 file support allows the user to create, maintain, and modify RT–11 file-structured disk devices. Volumes written with the MicroPower/Pascal ACP may be read by RT–11, VAX/VMS (using EXCHANGE), and RSX–11 (using FLX).

The network OPEN support allows the ACP to parse and create session control messages, required when using the NSP. If the NSP is not being used in your application, the code required to parse and generate these messages is not required.

The ACP pool area is used by the ACP in processing open requests. This area may need to be adjusted in size, depending on the number of NSP open requests that are currently in progress (180 bytes required per open) and the number of concurrently open disk files (40 bytes required per channel).

Figure 2–1: ACP Prefix File (ACPPFX.MAC) Excerpt

```
        .TITLE  ACPPFX - Ancillary Control Process prefix file
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-

        .MCALL macdf$
        macdf$

        RT$IPR == 250.          ; initialization priority
        RT$PPR == 175.          ; processing priority
        DIR$PR == 175.          ; directory operation priority

;+
; NSPSUP = (0 or 1)
;   0 = No NSP open support
;   1 = Include NSP open support
;-
        NSPSUP = 1 ; include NSP support
;+
; RTSUP = (0 or 1)
;   0 = No RT--11 Directory support
;   1 = Include RT--11 Directory support
;-
        RTSUP  = 1              ; include RT--11 directory support

;+
; ACP pool size in bytes
;-

$DPLSZ == 1000.                 ; 180. per NSP open
                                ; 40. per channel

;+
; END OF USER PARAMETERS
;-
        .end
```

2.7 Application Note: Device-Name Parsing

The ACP parser converts device names into SDBs. The device name in a file specification may be:

- A ring buffer name.
- A standard device name in the form ddcu, where dd is the 2-letter device name, c is an optional controller letter (default is A), and u is an optional unit number (default is 0). A dollar sign ($) is added to the beginning of the device name and controller letter to form the structure name (example: DUA0: is converted to '$DUA' unit 0).
- A logical name. A logical name should translate to one of the other forms described here.

Device strings are converted to uppercase before processing, so only uppercase kernel structures may be accessed via the OPEN statement. Any names less than six characters are padded with spaces to a 6-character length.

If no device name is specified, the ACP uses the default device name 'DK '. The user is assumed to have created a logical name for DK, either at build time in the kernel configuration file, or at run time via a call to the CREATE_LOGICAL_NAME routine.

2.8 FALACP

FALACP.PAS is a small version of the ACP, which you can use only in specific applications. FALACP performs terminal and ring buffer OPENs for a FALCON or KXT11-CA application having more than one serial line. This program opens files from 'TTA1:' to 'TTA9:', 'XLO0:' to 'XLO9', and 'XLI0:' to 'XLI9:'. For applications that require only terminal or ring-buffer access on the FALCON or KXT11-CA, you can use this program to replace the standard ACP.

Unlike the standard ACP, FALACP performs minimal device checking, making use of the first device name character to discriminate between terminal driver and ring buffer access. You are responsible for using the correct terminal line name or ring buffer name.

To use this alternate ACP, build FALACP into your application instead of the standard ACP driver. You do not need the ACPPFX.MAC prefix file. FALACP.PAS is in MICROPOWER$LIB for VMS and in LB:[2,10] for RSX.

Chapter 3
Asynchronous Serial Line (Terminal) Driver

This chapter describes the use of the MicroPower/Pascal asynchronous serial line (TT) driver, sometimes referred to as the terminal driver. The driver supports I/O operations on terminals and other devices attached to the following serial line interfaces:

- DLV11-type—DLV11, DLV11-E, DLV11-F, DLV11-J
- DLART-type—KXT11-CA or KXJ11-CA console, SBC-11/21, CMR21, MXV11-A, MXV11-B
- DZV11
- KXT11-CA or KXJ11-CA multiprotocol chip

The supported devices interface one or more asynchronous serial communication lines to a MicroPower/Pascal target processor for communication with terminals, modems, and other processors.

3.1 TT Driver Features and Capabilities

The TT driver supports read and write operations, the returning or altering of line parameters, and a stop function for outstanding read requests. All data transmissions use the same baud rate for sending and receiving. All lines run in 8-bit mode with one stop bit and no parity.

Read operations on a line are performed in line or block mode, as determined by prefix file default or a Set Characteristics operation.

In line mode, terminal-oriented line-editing operations, such as line erasure (CTRL/U), previous-character deletion (DELETE), and line redisplay (CTRL/R), are performed. Characters are echoed (if echo is enabled) as they are read. No data is returned to the requesting process until a carriage return is typed or the edit buffer overflows. The size of the edit buffer is specified in the TT driver prefix file.

In block mode, all data is passed to the requesting process without interpretation (unless XON/XOFF flow control is enabled). This allows you to connect the serial lines to devices other than terminals. For example, you can use the TT driver in conjunction with the asynchronous DDCMP (CS) driver to communicate with another MicroPower/Pascal target system over a serial line. See Chapter 12 for details.

In block mode, minimum/maximum read requests are honored. This allows your program—in particular, the OTS routines that carry out Pascal I/O procedure requests—to read a minimum number of bytes to complete your request plus as many other bytes (up to a maximum) as are immediately available. This facility is useful for high line-speed applications. Minimum/maximum read requests are possible, because the TT input ISR has two buffers and can buffer characters between reads. The size of the ISR input buffers is set in the TT driver prefix file.

Get and Set Characteristics functions allow the requesting process to inspect and change line parameters, including baud rate, modem status flags, input/output flow control (XON/XOFF), line/block mode, character length, even/odd parity, number of stop bits, and echo. Line parameters are initially set according to default values you specify in the TT driver prefix file.

The stop function allows the requesting process to reclaim resources by aborting an in-progress read request.

Modem control is supported for the DLV11-E, DHV11, and KXT11-CA or KXJ11-CA multiprotocol channel A interfaces and, in a limited fashion, for the DZV11 interface. Modems allow you to connect remote terminal lines to the serial line interface for access to the target processor. The modem is controlled by a set of signals it exchanges with the target processor. (More information on modem control signals is provided in Section 3.4.3.)

Modem control interrupts are supported for DLV11-E, DHV11, and KXT11-CA and KXJ11-CA multiprotocol channel A. The Set Modem Semaphore command allows the requesting process to specify a binary or counting semaphore to be signaled on each interrupt.

3.2 Performing Asynchronous Serial I/O

For most MicroPower/Pascal applications, you perform asynchronous serial I/O—particularly terminal I/O—by invoking Pascal I/O procedures that open files for terminal data and then input or output the data, in accordance with the rules for Pascal I/O. (INPUT and OUTPUT are opened implicitly and thus require no explicit OPEN invocation.) The Pascal I/O procedures—OPEN, GET, WRITE, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide.*

Note

The TT driver Set Modem Semaphore operation cannot be performed with Pascal I/O procedures. See Section 3.3 for more information on such operations.

In addition to invoking the Pascal I/O procedures, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the serial-line controller interrupt vector addresses
2. Edit the TT driver prefix file to reflect:
   - [For each controller:] Controller type, CSR address, interrupt vector address, hardware interrupt priority, and number of serial lines
   - [For each line:] ISR buffer size, speed, edit buffer size, and where supported by hardware, the setting or clearing of such parameters as input or output flow control (XON/XOFF), line editing (with or without echo of characters as they are read), bits/character, parity bits, number of stop bits, modem status-change interrupts, baud rate programming, Data Terminal Ready or Request to Send indications, or BREAK assertion

- Driver initialization and request-handling process priorities

3. Build into your application the following I/O system components:
   - TT driver process
   - [For explicit terminal file OPEN:] Ancillary control process (ACP)
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any terminal I/O support routines you opt to include (see kit files GETSET.PAS, GSINC.PAS, VT100.PAS, and VT1INC.PAS)

For more information on setting up your application software for terminal I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 3.6 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

When a module that contains Pascal I/O procedure invocations is built into your application, Pascal OTS routines for file service are linked to the module. The OTS file routines perform all Pascal operations on files, including file opening, input, and output. In particular, they perform the necessary low-level processing of high-level operations like OPEN and WRITE. Thus, the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to driver or ACP queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores—are concealed from the Pascal user.

Alternatives to using the Pascal I/O procedures for terminal I/O exist, but require more effort. You can:

- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and the driver, bypassing the OTS file routines (lower-level file system access)
- Issue your own Pascal or MACRO–11 packet-level requests to the driver, bypassing the OTS file routines and the ACP (nonfile access)

The following sections describe the Pascal I/O procedure interface to the TT driver, the lower-level request/reply packet interface, the status codes that can be returned to users of either interface, and the TT driver prefix file. An application note on hardware buffering concludes the chapter.

3.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O to an asynchronous serial line, you must open a file. Opening the file associates a Pascal file variable with a serial line unit. Invoke the OPEN procedure as follows:

```
OPEN (filvar, 'TTAu:', ...)
```

where:

- filvar is a Pascal file variable.
- u is a serial line number (0, 1, ...).

For example, 'TTA1:' would specify the second line (1) of the first serial interface controller listed in the TT driver prefix file.

Note

Any number of serial lines are supported, but the number is limited for each type of controller—up to four for DZV11, up to eight for DHV11, and one for most others. The range of valid identifying unit numbers is 0 through (n–1) for n lines configured in the TT driver prefix file. Lines are numbered sequentially upward from 0 in the order they appear in the prefix file, crossing controller boundaries.

The standard Pascal file variables INPUT and OUTPUT are implicitly associated (by default) with 'TTA0:'. They require no explicit OPEN invocations.

The OPEN statement causes the Pascal OTS to send an open request to the ACP, which returns a unit number and a TT driver request semaphore ID to the OTS. Subsequent I/O requests are sent directly to the driver by the OTS, with no further ACP involvement.

In carrying out subsequent input, output, CLOSE, or PURGE operations on serial interface units, the Pascal OTS uses the following packet-level driver functions:

- Read Logical (IF$RDL)
- Write Logical (IF$WTL)
- Close (IF$CLS)
- Purge (IF$PRG)

The appropriate request packets are sent to the driver only when necessary to complete a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the TT driver. Other Pascal statements unconditionally cause the OTS to issue send requests; examples are BREAK, which generates a Write Logical (IF$WTL), and CLOSE, which generates a Close (IF$CLS) request (normally preceded by a Write Logical, unless BREAK immediately precedes CLOSE).

Pascal Get and Set Characteristics functions that report or alter the characteristics or status of serial lines are provided in the file GETSET.PAS on the MicroPower/Pascal distribution kit. Those functions issue Get or Set Characteristics (IF$GET or IF$SET) request packets to the driver.

Neither the Set Modem Semaphore (IF$SMD) nor the Stop I/O (IF$STP) packet-level driver function can be performed with normal Pascal I/O statements or GETSET functions. To perform the Set Modem Semaphore or the Stop I/O function, either use the request/reply packet interface directly or write Pascal procedures that take a user-specified file variable (or queue semaphore ID) and send the appropriate request packet to the driver. (The Get/Set Characteristics procedures in GETSET.PAS demonstrate the latter approach.)

Note

Pascal procedures for manipulating VT100 video are distributed as source modules on the MicroPower/Pascal kit. The relevant files are VT100.PAS, which contains the procedures, and the include file VT1INC.PAS, which externally declares the procedures. Most of the operations WRITE to OUTPUT.

3.4 Request/Reply Packet Interface

The following packet-level functions provided by the TT driver are listed by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical
IF$RDL (1)	Read Logical
IF$WTP (3)	Write Physical
IF$WTL (4)	Write Logical
IF$SET (6)	Set Characteristics
IF$GET (7)	Get Characteristics
IF$STP (10)	Stop I/O
IF$SMD (11)	Set Modem Semaphore

If a request is received for an Open (IF$LOK or IF$ENT), a Close (IF$CLS), or a Purge (IF$PRG), the driver returns an illegal function status code (ES$IFN), which the ACP (Open) or OTS (Close/Purge) interprets as indicating that no device-dependent processing was required for that operation.

Note

The MACRO–11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The following function modifiers recognized by the TT driver are shown listed by symbolic code and bit position:

Code	Function
FM$MIN (bit 7)	Enable minimum/maximum block-mode read
FM$BSM (bit 13)	Signal binary/counting semaphore

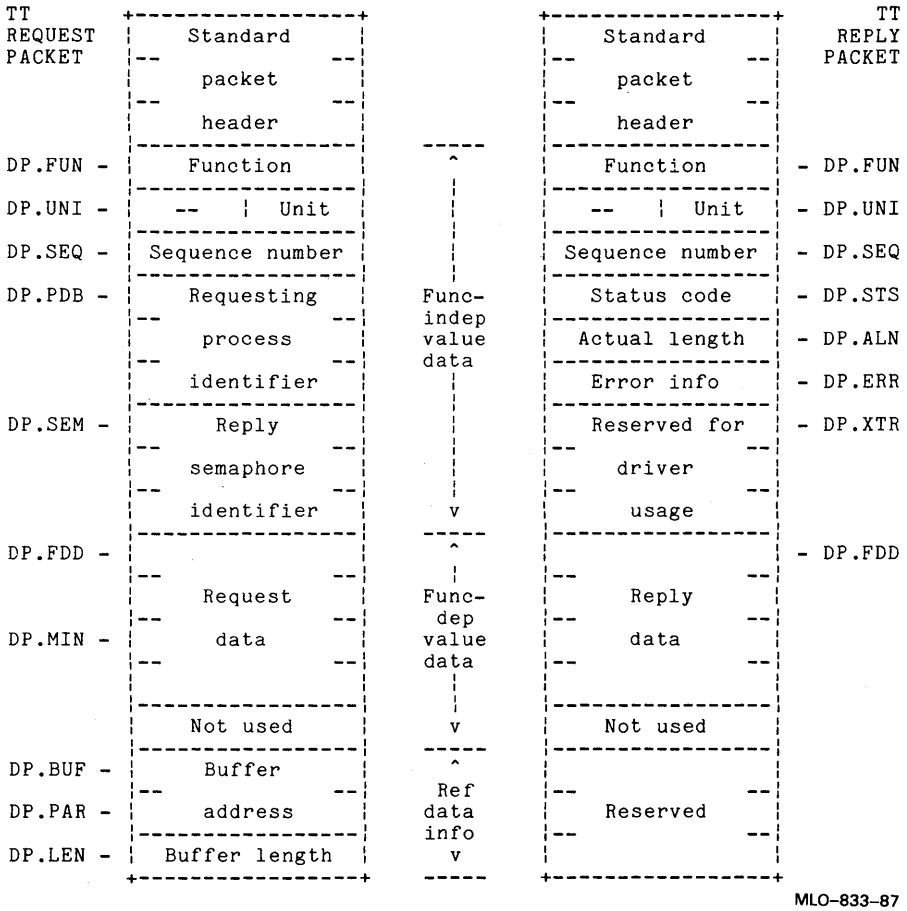
The TT driver is a single (static) process, beginning as an initialization process and then lowering its priority to the running level specified in the prefix file. The single process handles all the controllers (serial interface units) and lines specified in the prefix file, unlike other MicroPower/Pascal drivers that create a separate process for each controller. I/O requests for any controller are sent (using a Pascal SEND or a MACRO–11 SEND$) to the request queue semaphore waited on by the driver process.

The request queue name and number of supported units for TT driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
Asynchronous serial	$TTA	1–n (1–4 for DZV11, 1–8 for DHV11, 1 for most others)	0 through (n–1) in prefix file order, crossing controller boundaries

The units configured for each controller must be specified in the TT driver prefix file.

The general format of the TT driver request and reply packets follows:



The function-independent portions of the packets are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets for each type of TT driver function are described in the following sections.

Note

The MACRO–11 field names shown do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

3.4.1 Read Functions

When a read (IF$RDP or IF$RDL) request is received, the TT driver validates the request and queues it on the specified line. If no request is currently active, the operation is begun.

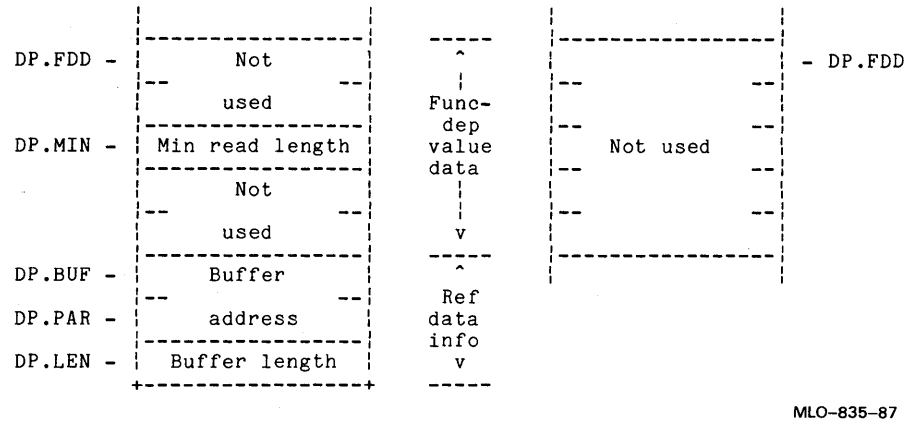
Reads are performed in block mode, unless you enabled line editing for the line in question in the prefix file or in a Set Characteristics request.

In line mode, line-editing functions are performed with optional echoing of characters as they are read. No data is returned to the requesting process until a carriage return has been entered, regardless of the requested read length. Thus, even a single-character request must wait for a carriage return—unless a portion of a previously entered line remained in the line buffer when the operation commenced.

In block mode, the request is checked for the minimum/maximum (FM$MIN) function modifier and a minimum read value (offset DP.MIN in the request packet). If both are present, the value specified at offset DP.MIN in the request packet is used as the required read size; if either is absent, the reference buffer length (DP.LEN) is used as the required size. Once the required amount of data has been received, the request is considered complete. If FM$MIN was specified, up to (maximum-minimum) additional bytes of data will be returned to the user if they are currently available in the ISR buffers. The request is then returned to the user with the actual-length field (offset DP.ALN), reflecting the actual length of the transfer.

If input flow control is enabled for the line (by prefix file default or Set Characteristics request), the input ISR sends XOFFs to the device attached to the line whenever the input ISR buffer is 75% full. When the congestion is reduced, an XON is sent to allow further input.

The function-dependent portions of the read request and reply packets are shown below:



MLO-835-87

Fields DP.BUF through DP.LEN specify the location and length of the user buffer that is to receive the data. Those fields are put into the packet by the kernel send primitive, based on the send call arguments.

The DP.MIN field can be used to specify a minimum read length for block-mode reads. If function-modifier FM$MIN is set, the number of bytes returned by a block-mode read is the amount specified in DP.MIN plus as many bytes, up to the DP.LEN maximum, as were available in the input ISR buffers when the minimum length was achieved. If DP.MIN is zero, this has the effect of a conditional read. DP.MIN is ignored for line-mode reads.

If FM$MIN is not set, DP.LEN is used as the required read length.

In line mode, the length specified at DP.LEN is honored, but regardless of the number of available bytes, no data is returned until a terminator has been entered.

3.4.2 Write Functions

When a write (IF$WTP or IF$WTL) request is received, the TT driver validates the request and queues it on the specified line. If no request is currently active, the operation is begun.

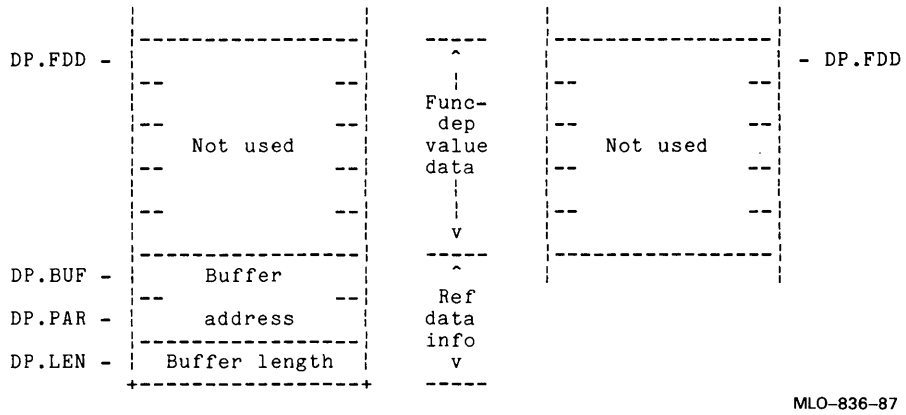
Note

Write requests have priority over pending echo for output. Thus, if a user application performs continuous writes, pending echo may be delayed indefinitely.

If output flow control is enabled for the line (by prefix file default or Set Characteristics request), XOFFs received from the device attached to the line suspend output, until an XON is received.

Replies to write requests are not sent to the caller until all data has been given to the device. Generally, this means that all data except the last two bytes has been transmitted. If the application requires complete output synchronization, it writes one or two null bytes. See the application note on hardware buffering at the end of this chapter for details.

The function-dependent portions of the write request and reply packets follow:



MLO-836-87

Fields DP.BUF through DP.LEN specify the location and size of the user buffer from which data is to be copied. Those fields are put into the packet by the kernel send primitive, based on the send call arguments.

3.4.3 Get and Set Characteristics Functions

The Get Characteristics (IF$GET) and Set Characteristics (IF$SET) functions allow you to inspect or change the current parameters of a given line. The parameters include bit settings for:

- Input/Output flow control (XON/XOFF)
- Line/Block mode
- Echo (line mode only; characters are echoed as they are read)
- Read-only modem controls—Ring, Carrier, Clear to Send, Data Set Ready (for DLV11-E, DHV11, KXT11-CA or KXJ11-CA multiprotocol channel A; Ring and Carrier only for DZV11)
- Read/write modem controls—Data Terminal Ready, Request to Send, Enable Modem Interrupts (for DLV11-E, DHV11, KXT11-CA or KXJ11-CA multiprotocol channel A; DTR only for DZV11)
- Assert/Deassert BREAK
- Programmable baud rate (only for DLV11-E, DLV11-F, DLART, KXT11-CA or KXJ11-CA multiprotocol, DHV11, DZV11)
- Setting the line's framing characteristics: bits/character, parity, stop bits
- Terminal type

Note

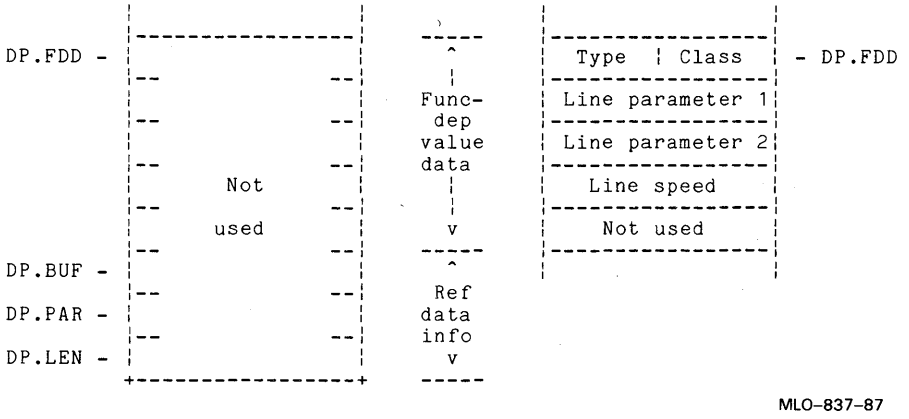
No modem control is provided for KXT11-CA or KXJ11-CA multiprotocol channel B. Channel A can be configured with full modem control or no modem control. The list above assumes full modem control for channel A.

Split line speeds are not supported; a line's transmit and receive speeds must match.

When a Get Characteristics request is received, the TT driver gets the line status settings from the transmit and receiver CSRs and from its internal control block for the specified line and passes those parameters back to the requesting process.

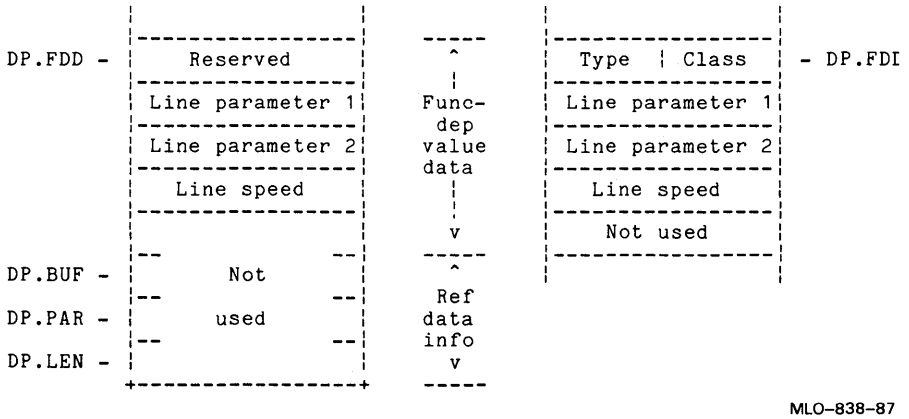
When a Set Characteristics request is received, the TT driver sets or clears bits in the transmitter and receiver CSRs and in its internal control block for the specified line and then performs a Get Characteristics operation, which passes the new line parameters back to the requesting process.

The function-dependent portions of the Get Characteristics request and reply packets follow:



MLO-837-87

The function-dependent portions of the Set Characteristics request and reply packets are shown below:



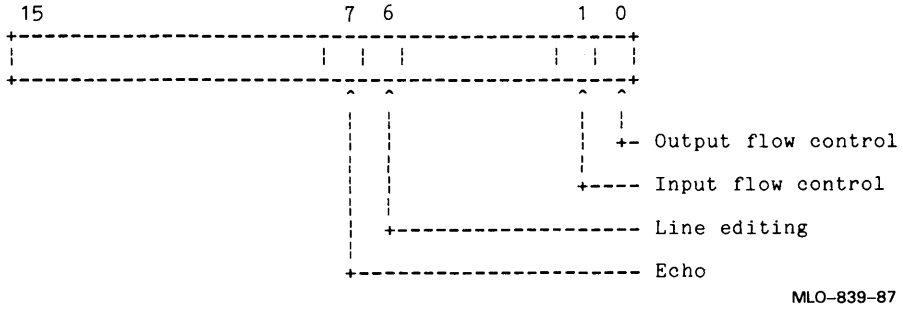
MLO-838-87

Device class and type information is returned at offsets DP.FDD and DP.FDD+1 in the Get and Set Characteristics reply packets. In those fields:

- Class is DC$TER for asynchronous serial line interface.
- Type indicates the specific type of interface:
  TT$DL for minimum serial line capability (DLV11, DLV11-J, MXV11-A)
  TT$DLE for DLV11-E
  TT$DLF for DLV11-F
  TT$DLT for DLART (SBC-11/21, MXV11-B, KXT11-CA console, CMR21)
  TT$DLU for DLART (KXJ11-CA console)
  TT$DM for KXT11-CA or KXJ11-CA multiprotocol, data line only port
  TT$DMM for KXT11-CA or KXJ11-CA multiprotocol with modem control
  TT$DH for DHV11
  TT$DZ for DZV11

The first and second line parameters (at DP.FDD+2 and DP.FDD+4 in the packets just shown) are identical to the parm1 and parm2 arguments used in calls to the TTLIN$ prefix file macro. (See Section 3.6.) The TT line parameters select the characteristics to be set or report the current line characteristics.

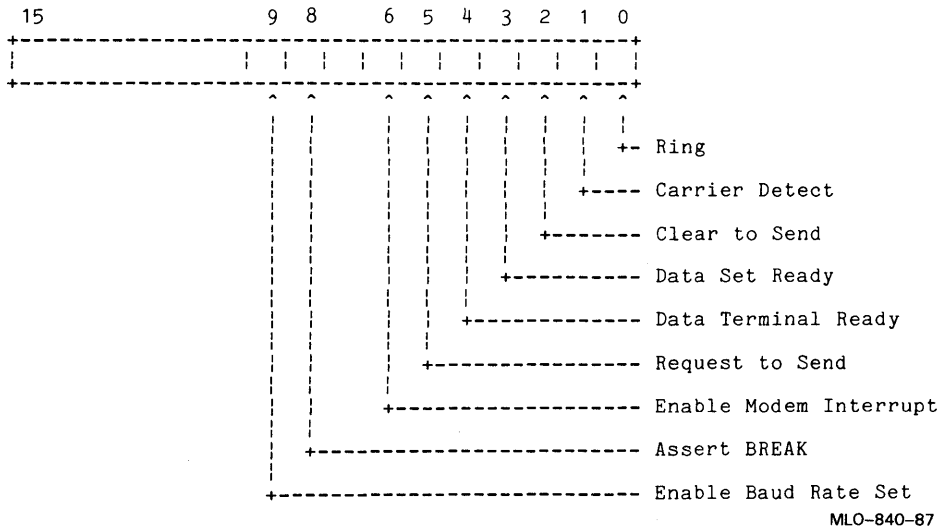
The format of the first line parameter is shown below:



The four bits labeled above correspond to the TTLIN$ C.xxxx options:

- Bit 0, if set, enables output flow control (XON/XOFF).
- Bit 1, if set, enables input flow control (XON/XOFF).
- Bit 6, if set, enables line editing (line mode for read operations).
- Bit 7, if set, enables echo of characters as they are read, provided the line-editing bit (6) is also set.

The format of the second line parameter is shown below:



Bits 0 (Ring) through 3 (Data Set Ready) are read-only. The remaining labeled bits correspond to the TTLIN$ E.xxx options. Bits 0 (Ring) through 6 (Enable modem status-change interrupts) are modem control bits. Proceeding from right to left in the format above:

- Bit 0, if set, indicates a Ring, informing the target processor that an incoming call signal is being received by the modem.
- Bit 1, if set, indicates Carrier Detect, informing the target processor that the data channel signal is OK, receiver is ready.
- Bit 2, if set, indicates Clear to Send, informing the target processor that the modem is ready to transmit data.
- Bit 3, if set, indicates Data Set Ready, informing the target processor that the modem is in data mode and ready to operate.
- Bit 4, if set, indicates Data Terminal Ready, informing the modem that the target processor is ready to transmit or receive data; if clear, the modem disconnects itself from the line.
- Bit 5, if set, indicates Request to Send, telling modem to enter transmission mode; if clear, the modem leaves transmission mode after data transmission.
- Bit 6, if set, enables modem status-change interrupts (only for DLV11–E, KXT11–CA, or KXJ11–CA multiprotocol with full modem control, or DHV11).
- Bit 8, if set, asserts a BREAK (must be cleared by software).
- Bit 9, if set, enables software-setting of the baud rate specified in the TT line speed parameter. (Device must be jumpered to allow programmable baud rate.)

- Bits 10 and 11, select character length as follows:

Setting	Length
00	= 5
01	= 6
10	= 7
11	= 8

- Bit 12, if set, generates parity bit for each character. If clear, no parity bits are generated.
- Bit 13, if set, generates even parity. If clear, odd parity is generated. This bit has no effect if bit 12 is clear.
- Bit 14, if set, generates two stop bits rather than one. (If you have selected a character length of 5 and you select two stop bits, 1.5 stop bits are generated for each character.) If clear, one stop bit is generated for each character.
- Bit 15, if set, modifies the line's framing characteristics through use of the values in bits 10–14. If clear, bits 10–14 have no effect on the line's framing characteristics.

With a KXT11-CA/KXJ11-CA multiprotocol chip, if you have selected 5-bit mode, the three high-order bits of each data byte must be 0, or unpredictable errors occur.

The line speed parameter (at offset DP.FDD+6) contains a value that sets the baud rate—provided the device is jumpered to allow software programming of baud rate and bit 9 of the second line parameter is set. In a TT Get/Set Characteristics reply packet, the speed parameter gives the current baud rate.

The following shows possible decimal line speed values:

Value	Baud	Notes
1	50	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
2	75	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
3	110	Invalid for DLART
4	134.5	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
5	150	Invalid for DLART
6	200	Valid only for DLV11 type
7	300	
8	600	
9	1200	
10	1800	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
11	2000	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
12	2400	
13	3600	Invalid for DLART, KXT/KXJ multiprotocol, DHV11

Value	Baud	Notes
14	4800	
15	7200	Invalid for DLART, KXT11-CA/KXJ11-CA multiprotocol
16	9600	
17	19200	Invalid for DZV11
18	38400	Invalid for DLV11-E/F, DZV11
19	76800	Valid only for KXT11-CA/KXJ11-CA multiprotocol

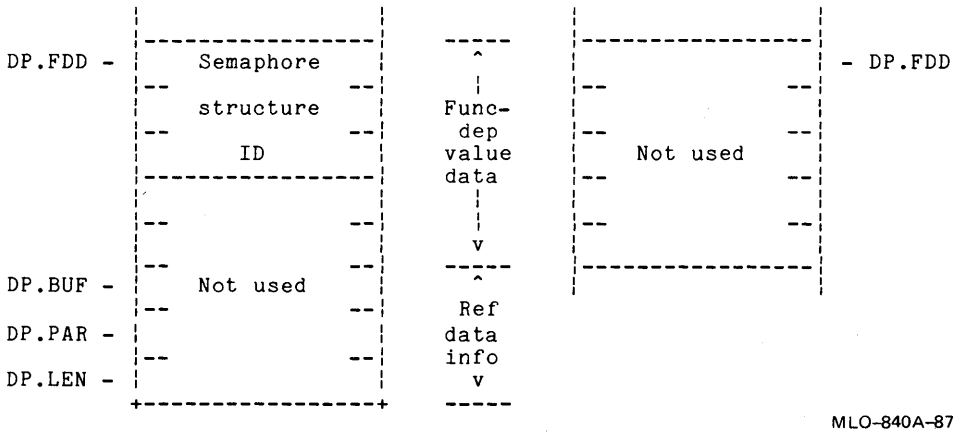
Note

For DHV11 line-pairs, two sets of possible baud rates (A and B) are listed in the DHV11 hardware guide. When selecting DHV11 baud rates, remember that both members of a line-pair must use baud rates from the same set.

3.4.4 Set Modem Semaphore Function

The Set Modem Semaphore (IF$SMD) function is used to specify the binary or counting semaphore to be signaled at each modem interrupt. Modem interrupts are generated when a change in modem status occurs on a specified line. After issuing this command, you would normally send a Set Characteristics command, enabling modem status interrupts. Modem interrupts are supported only by DHV11, KXT11-CA or KXJ11-CA multiprotocol channel A, and DLV11-E hardware. To disable modem status signaling, you can send a set command disabling modem interrupts. To change semaphores, you can send another Set Modem Semaphore command specifying a different semaphore.

The following shows the function-dependent portions of the Set Modem Semaphore request and reply packets:



The binary or counting semaphore specified at offset DP.FDD is placed in the TT driver's internal control block for the line specified at offset DP.UNI (function-independent portion). The specified semaphore is signaled whenever a modem control interrupt occurs on a DLV11-E, DHV11, or KXT11-CA or KXJ11-CA multiprotocol channel A.

The calling program is responsible for issuing a Get Characteristics request to determine the status on each signal and for taking appropriate action (possibly including a Set Characteristics operation). The file GETSET.PAS on the MicroPower/Pascal distribution kit provides a model for getting and changing characteristics.

3.4.5 Stop Request

The Stop Request (IF$STP) function lets you stop an in-progress read. DP.ALN of the read reply packet contains the number of bytes already transferred to your buffer at the time the terminal driver begins processing the stop request.

For lines in edit mode, your buffer gets filled with the contents of the edit buffer at the time the stop request is processed. The number of characters transferred are MIN (characters in edit buffer, DP.LEN). After your buffer has been filled, any additional characters in the EDIT buffer are flushed.

DP.STS of the read reply packet contains ES$ABO.

The stop request is returned with a status of ES$NOR. It is returned by the driver after the stopped read request is returned.

If you have an outstanding Pascal read request (as opposed to packet level I/O) and a stop I/O is issued, the OTS raises an ES$ABO exception for the process that issued the read. You should be prepared to handle the exception that occurs as a result of the stopping of the read.

Characters that arrive on the line while the stop request is being processed are buffered and are available for the next read request on that line. These characters are placed in an internal buffer different from the edit buffer. No characters are transferred from the internal buffer to the edit buffer or to the user buffer while the stop request is being processed.

If you issue a stop request for a line on which no read request is currently in progress, the driver returns an ES$NIP (no I/O in progress) exception.

3.5 Status Codes

If a serial interface device or the TT driver detects an error during an I/O operation, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error was detected during the I/O operation, the driver returns a value of ES$NOR (0) in the status-code (DP.STS) field of the reply message.

The TT driver returns the following exception codes:

Code	Type	Description
ES$FRM	HARD_IO	Framing error
ES$IVP	HARD_IO	Invalid parameter: software set of baud rate not allowed for this device, baud rate illegal for this device
ES$NXU	HARD_IO	Nonexistent unit: invalid unit number
ES$OVF	HARD_IO	Data (software buffer) overflow
ES$OVR	HARD_IO	Device overrun
ES$PAR	HARD_IO	Parity error
ES$IFN	SOFT_IO	Illegal function code; also used internally to signal ACP or OTS that no device-dependent processing of an Open, Close, or Purge was required
ES$NRF	SOFT_IO	No reference data present for read or write request
ES$ABO	SOFT_IO	Read request aborted
ES$NIP	SOFT_IO	No I/O in progress for specified line

Exception codes are defined in the EXC.PAS include file for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for OTS-detected I/O errors or for kernel-detected errors that the TT driver raises rather than passing back to the requesting process. OTS-detected I/O errors are listed in Chapter 9 of the *MicroPower/Pascal Language Guide*.

3.6 TT Driver Prefix File

The TT driver prefix module is distributed in four versions—TTPFX.MAC, TTPFXC.MAC (CMR21 version), TTPFXF.MAC (SBC-11/21 version), and TTPFXK.MAC (KXT11-CA and KXJ11-CA versions). The versions differ only in their selection of the default (uncommented) macro calls for a particular board.

Figure 3-1 shows TTPFX.MAC. The following paragraphs describe the macro calls and symbol definitions that can be edited to fit your application.

The TTCTR$ macro is invoked once for each controller serviced by the driver. Its parameters are device type, CSR address, interrupt vector, hardware priority, and number of lines.

Note

The interrupt vector supplied in the prefix file is the receive-side vector for a given controller; the transmit vector is assumed to follow the receive vector by 4 bytes. For example, vec=300 implies a corresponding transmit vector at location 304. Both vectors would be specified in the DEVICES macro in the system configuration file—for example, "DEVICES ... 300, 304."

The possible device types are:

- TT$DL for minimum serial line capability (DLV11, DLV11-J, MXV11-A)
- TT$DLE for DLV11-E
- TT$DLF for DLV11-F
- TT$DLT for DLART (SBC-11/21, MXV11-B, KXT11-CA console, CMR21)
- TT$DLU for DLART (KXJ11-CA console)
- TT$DM for KXT11-CA or KXJ11-CA multiprotocol, data line only port
- TT$DMM for KXT11-CA or KXJ11-CA multiprotocol with full modem control
- TT$DH for DHV11
- TT$DZ for DZV11

The TTLIN$ macro is invoked once for each configured line. Its parameters are ISR buffer size, two parameters (parm1 and parm2) of status bit-settings, line speed, and edit buffer size.

The options for TTLIN$ parameters parm1 and parm2, described below, correspond to bit settings in the TT driver Set Characteristics request packet. The Set Characteristics request can be used to change line characteristics at run time.

For lines that are to be used by the asynchronous DDCMP (CS) driver for DDCMP message exchange, you must not enable flow control (XON/XOFF) or line editing. See Chapter 12 for details.

Note

For serial hardware in which each line is associated with its own CSR/vector pair, the TTCTR$ and TTLIN$ macros are invoked in pairs for each line. For example, the DLV11-J is considered a single controller in the hardware sense. However, each DLV11-J line, by virtue of being associated with a unique CSR/vector pair, is considered a separate controller by the MicroPower/Pascal software. So the controller and line macros, TTCTR$ and TTLIN$, must be invoked in pairs for each DLV11-J line.

The following TT$IPR and TT$PPR definitions determine the priority at which the TT driver process initializes and the priority to which it lowers itself for request processing. Note that no xx$HPR hardware priority symbol appears. The TT driver, unlike most other standard MicroPower/Pascal drivers, services several different types of controllers under the umbrella of a single process. Thus, a different hardware priority is specified—in a TTCTR$ call—for each controller.

Figure 3–1: TT Driver Prefix File (TTPFX.MAC)

```
        .NLIST
        .ENABL  LC
        .LIST
        .TITLE  TTPFX - Terminal/Serial Line Driver Prefix file
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
        .mcall   macdf$, drvdf$, ttpfx$
        macdf$
        drvdf$
        ttpfx$
;+
; Define globals symbols needed for the TT process
;-
TT$IPR  ==      250.            ;Initialization priority
TT$PPR  ==      175.            ;Normal process priority
;+
; This is where the user defines the asynchronous lines.
; TTCTR$ is used to define the device controller, TTLIN$
; defines each of the lines associated with the controller.
; TTLIN$(s) must follow (immediately) its (their) TTCTR$ definition.

;
; The order of the TTLIN$ defines the unit numbers.  Thus
; the first TTLIN$ is unit 0, the second unit 1, etc....
;
; Options for parm1 are:
;   C.OFLW      enable output flow control (terminal/host XON/XOFF)
;   C.IFLW      enable input flow control (host/terminal XON/XOFF)
;   C.LINE      enable line editing
;   C.ECHO      If C.LINE has been selected, enable echo of characters
;               as they are read.
;


; Options for parm2 are:
;   E.DTR       Set Data Terminal ready (DTR)
;   E.RTS       Set Request to send
;   E.DIE       Enable modem interrupts (TT$DLE, TT$DMM, TT$DH)
;   E.BRK       Set Break (must be cleared by software)
;   E.PBD       Software set selected baud rate.  This option
;               should only be used if the device is jumpered
;               to allow software programming of the baud rate.
;-

; DLV-11 Console SLU
; WARNING: Do not define this line for applications with PASDBG support
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO      ; Full XON/XOFF, ECHO, LINE
;       foo1 = 0                        ; Use jumpered/default baud rate
;       ttctr$ type=TT$DL, csr=177560, vector=60, hprio=4, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
```

```
; DLV-11 SLU2
        foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
        foo1 = 0                          ; Use jumpered/default baud rate
        ttctr$ type=TT$DL, csr=176500, vector=300, hprio=4, nlines=1
        ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; KXT11--CA/FALCON/CMR21 Console DLART
; WARNING: Do not define this line for applications with PASDBG support
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO      ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD                            ; Set programmed baud rate
;       ttctr$ type=TT$DLT, csr=177560, vector=60, hprio=4, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; KXJ11--CA Console DLART
; WARNING: Do not define this line for applications with PASDBG support
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO      ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD                            ; Set programmed baud rate
;       ttctr$ type=TT$DLU, csr=177560, vector=60, hprio=4, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; FALCON SLU2 DLART  ( NOTE: hprio=5 for SLU2 )
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD                            ; Set programmed baud rate
;       ttctr$ type=TT$DLT, csr=176540, vector=120, hprio=5, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; KXT11--CA/KXJ11--CA Multiprotocol channel A (SLU2A) with modem control
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD!E.DTR                      ; Set baud rate & DTR
;       ttctr$ type=TT$DMM, csr=175700, vector=140, hprio=4, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; KXT11--CA/KXJ11--CA Multiprotocol channel B (SLU2B) (Note: Channel B has no
; modem control)
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD                            ; Set programmed baud rate
;       ttctr$ type=TT$DM, csr=175710, vector=160, hprio=4, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; CMR21 Port 3 (Note Hardware priority = 5)
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD                            ; Set programmed baud rate
;       ttctr$ type=TT$DLT, csr=175620, vector=124, hprio=5, nlines=1
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

; DZV-11
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO       ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD!E.DTR                      ; Set baud rate & DTR
;       ttctr$ type=TT$DZ, csr=160100, vector=310, hprio=4, nlines=4
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
```

```
; DHV11
;       foo  = C.OFLW!C.IFLW!C.LINE!C.ECHO           ; Full XON/XOFF, ECHO, LINE
;       foo1 = E.PBD!E.DTR                           ; Set baud rate & DTR
;       ttctr$ type=TT$DH, csr=160020, vector=320, hprio=4, nlines=8.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.
;       ttlin$ ibuf=20, parm1=foo, parm2=foo1, speed=9600, edtbuf=80.

        ttfin$          ; Finish up after generating the data structures
        .end
```

3.7 Application Note: Hardware Buffering

TT driver packet-level write requests are not replied to the caller until all data has been given to the device. Generally this means that all the data except the last two bytes has been transmitted. If the application requires complete output synchronization—a guarantee that all data has left the particular serial interface board—it writes one or two null bytes.

The relevant hardware buffering information is given below for each type of serial line controller:

Controller	Buffering
DLV11-J	Double-buffered input, double-buffered output; two null bytes should be written to guarantee all data has left the board.
DLV11, DLV11-E, DLV11-F, DLART	Double-buffered input, single-buffered output; one null byte should be written to guarantee all data has left the board.
DHV11	256-character input buffer, DMA output; all data has left the DUART; only one null byte required to guarantee all data has left the board.
DZV11	64-character input buffer, single-buffer output; one null byte should be written to guarantee all data has left the board.
KXT11-CA or KXJ11-CA multiprotocol chip	Quadruple-buffered input, double-buffered output; two null bytes should be written to guarantee all data has left the chip.

Chapter 4
Disk-Class Device Drivers

This chapter describes the use of the MicroPower/Pascal disk-class device drivers, which support I/O operations both on disks and on nondisk media that are treated as disks. The disk drivers support the mass-storage controllers, media, and protocols listed below:

Driver	Supported Controllers, Media, and Protocols
DL	RLV11 controller, RL01 disk (16/18-bit addressing) RLV12 controller, RL01/RL02 disks (16/18/22-bit addressing) RLV21 controller, RL01/RL02 disks (16/18-bit addressing)
DY	RXV21 controller, RX02 flexible diskettes (single/double density, 18-bit addressing)
DU	Mass Storage Control Protocol (MSCP) controllers and disks, including RQDX1, RQDX2, and RQDX3 controllers and RX50, RD51, RD52, RD53, and RC25 disks (22-bit Q-bus environment)
XD	Extended (> 65536 blocks) physical disks, partitioned for Pascal I/O
DD	TU58 DECtape II connected to DLV or KXT11-CA/KXJ11-CA serial line interface unit
VM	Virtual memory (mapped systems only, requires MMU)

The devices listed above provide mass storage for MicroPower/Pascal target applications.

Note

MSCP is a high-level interface to a family of devices and mass-storage controllers manufactured by DIGITAL.

4.1 Disk Driver Features and Capabilities

The disk-class drivers support read and write operations and the returning of device characteristics.

Logical read or write operations transfer data to or from a buffer in the calling process, starting at a disk address that is specified (at packet-level) in units of numbered, 512-byte logical blocks.

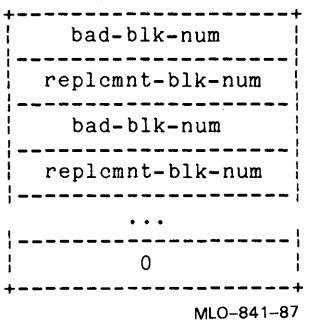
All RL01/2, RX02, and MSCP read and write operations use direct-memory-access (DMA) transfers; TU58 read and write operations use byte transfers; virtual memory read and write operations use word move (MOV) instructions.

Get Characteristics operations report standard device characteristics, including the storage capacity per disk unit (or XD partition) in terms of logical blocks.

In addition to logical read and write and Get Characteristics functions, most disk drivers support operations that are specific to the controllers, media, or protocols they support.

The RL01/2, RX02, and TU58 drivers support physical read and write operations, which specify the initial disk or tape address in terms of a track, cylinder, and sector (RL01/2, RX02) or a 128-byte physical record (TU58).

The RL01/2 driver supports bad-block replacement, using the manufacturer's bad-block replacement table, which resides in block 1 of the RL01 or RL02 disk. The table starts at the first word of block 1 and has the following form:



MLO-841-87

The bad-blk-num value is the logical block number of the bad block. The replcmnt-blk-num value is the logical block number of the replacement block. Replacement blocks reside on the disk's last track—second recording surface, last cylinder. The range of logical block numbers on the last track is 10220 to 10239 for the RL01 and 20460 to 20479 for the RL02. Logical blocks on the last track are write protected from access by logical block number; you can access the replacement area only by physical address. No more than 10 bad blocks are allowed per disk.

The RL01/2 driver also supports dynamic mounting and dismounting of disk packs. The driver detects when a new pack has been mounted and reads in a new copy of the bad-block replacement table. (Control of the mounting of the pack by an operator is the responsibility of the application program.)

The RX02 driver supports initialization (formatting) of a diskette for single- or double-density operation.

The TU58 driver supports read-with-increased-threshold and write-verify options. The extended disk driver supports the partitioning (subdividing) of disks with greater than 65,536 blocks so that Pascal file operations can be performed on them. The XD driver helps you overcome two current limitations on I/O to extended disks:

- The RT–11 file system's 16-bit orientation, which imposes a 65,536 block limit on an RT–11 directory-structured disk
- The current OTS and ACP interaction, in which 16 bits are used for I/O transfer computations with no allowances made for media with block counts that cannot be contained in 16 bits; this limitation imposes a 65,536 block limit on a non-directory-structured disk (for example, a disk opened as 'DUA0:')

The XD driver allows you to subdivide an extended disk into multiple partitions of up to 65,536 blocks each. You can then OPEN a partition or a named directory file in a partition as if the partition was a disk itself and perform normal Pascal file operations. (Section 4.3 gives the OPEN syntax for a named directory file or a non-directory-structured disk unit.)

Conceptually, the XD driver resides "between" the ACP and a physical disk driver (DL, DY, DU), receiving ACP requests for I/O and translating them into physical-disk driver requests. In the standard MicroPower/Pascal controller/unit terminology, each physical disk partitioned by the XD driver is considered a single controller and each partition a unit. Thus, an RD53 configured as DUA0: (for example) in the DU prefix file could be subdivided into three partitions that could be referenced in Pascal OPEN statements as 'XDA0:', 'XDA1:', and 'XDA2:'. According to that mapping, an I/O request for XDA1: would access the second partition of DUA0:.

Note

Another approach to extended disk I/O is to issue send requests directly to the physical disk driver, bypassing the OTS file routines, the ACP, and the XD driver (nonfile access).

4.2 Performing Disk I/O

For most MicroPower/Pascal applications, you perform disk I/O by invoking Pascal I/O procedures that open files for disk data and then input or output the data, in accordance with the rules for Pascal I/O. If a file is a named file on a directory-structured disk, you can also invoke Pascal I/O procedures that initialize the directory or rename, protect, or delete a file. If the disk is an RX02, you can invoke a Pascal I/O procedure that formats the disk for single- or double-density operation. Pascal I/O procedures—OPEN, GET, WRITE, INIT_DIRECTORY, DELETE_FILE, FORMAT_RX02, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide*.

Note

The disk driver physical read and write operations cannot be performed with Pascal I/O procedures. See Section 4.3.

In addition to invoking the Pascal I/O procedures, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the disk controller interrupt vector addresses (not applicable for the VM driver)

2. Edit the disk driver prefix file to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), number of controller units and their identifying numbers (0, 1, ...)
   - [For each nonVM controller:] CSR address and interrupt vector address
   - [For each VM controller (region):] Size of the memory region in 512-byte blocks
   - [For each DD controller:] Serial line type and speed
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. [For extended disk partitioning:] Perform steps 1 and 2 to configure the physical disk device driver(s) for the disks to be partitioned; then edit the XD driver source, XDDRV.PAS, to reflect:
   - Maximum number of disk blocks per partition (up to 65,536)
   - Minimum number of disk blocks per partition
   - Number of physical disks to be partitioned
   - [For each physical disk:] Request queue semaphore name and unit number associated with the physical disk (see Step 2) and XD request queue semaphore name; for each physical disk after the first, increase DATA_SPACE attribute by 456
4. Edit the ACP prefix file to indicate whether RT-11 directory support is required; the default is inclusion of directory support
5. Build into your application the following I/O system components:
   - Disk driver process
   - [For extended disk partitioning:] XD driver, as a user static process (NOT as a system process, as for other drivers); see Appendix B of the system user's guides for build details
   - [For disk file OPEN:] Ancillary control process (ACP)
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any disk I/O support routines you opt to include (see kit files FSPAS.PAS, INTDIR.PAS, GETSET.PAS, and GSINC.PAS)

For more information on setting up your application software for disk I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Sections 4.7 and 4.8 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

When a module that contains Pascal I/O procedure invocations is built into your application, Pascal OTS routines for file service are linked to the module. The OTS file routines perform all Pascal operations on files, including file opening, input, and output. In particular, they perform the necessary low-level processing of high-level operations such as OPEN and WRITE. Thus, the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to driver or ACP

queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores—are concealed from the Pascal user.

Alternatives to using the Pascal I/O procedures for disk I/O exist, but require more effort. You can:

- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and the driver, bypassing the OTS file routines (lower-level file system access).
- Issue your own Pascal or MACRO–11 packet-level requests to the driver, bypassing the OTS file routines and the ACP (nonfile access).

The following sections describe the Pascal I/O procedure interface to the disk drivers, the lower-level request/reply packet interface, status codes that can be returned to users of either interface, extended error information that the DL, DY, and DD drivers return to packet-level users, and disk driver prefix files.

4.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O to a disk, you must open a file. Opening the file associates a Pascal file variable with a named directory file or a non-directory–structured disk unit. For a named directory file, invoke the OPEN procedure with:

```
OPEN (filvar, 'ddcu:filnam.typ', ...)
```

where:

- filvar is a Pascal file variable.
- dd is the driver identifier (DL for RL01/2, DY for RX02, DU for MSCP, DD for TU58, VM for virtual memory, XD for extended disk).
- c is a controller identifier (A, B, ...; default is A).
- u is a controller unit number (0, 1, ...; default is 0).
- filnam.typ is the directory file name.

For a non-directory-structured disk file, invoke the OPEN procedure with:

```
OPEN (filvar, 'ddcu:', ...)
```

where filvar, dd, c, and u are the same syntactic elements described above. For example, 'DYA0:' would specify the first unit (0) of the first RX02 controller (A) listed in the DY driver prefix file.

The number of units supported for each disk-class controller follows:

Controller	Number of Units	Numbering
RL01/2	1–4 (any combination of RL01s and RL02s)	In prefix file
RX02	1–2	0 for left drive and 1 for right in dual-drive
MSCP	1–n	In prefix file
Extended disk	1–n (partitions), as determined by physical disk size and user-defined partition size	0 through (n–1)
TU58	1–2	0 for left drive and 1 for right in dual-drive
Virtual memory	1	0

The number of units configured for each controller and their unit numbers must be specified in a disk driver prefix file. Typically, unit numbering starts at 0.

The OPEN statement causes the Pascal OTS to send an open request to the ACP, which returns a channel number and an ACP request semaphore ID to the OTS. That information is used in subsequent Pascal I/O operations on the unit.

In carrying out subsequent input, output, CLOSE, PURGE, rename, delete, protect, and unprotect operations on disk units and files, the Pascal OTS and the ACP use the following packet-level driver functions:

- Read Logical (IF$RDL)
- Write Physical (IF$RDP)—for RX02 formatting
- Write Logical (IF$WTL)
- Rename (IF$REN)—directory files only
- Delete (IF$DEL)—directory files only
- Close (IF$CLS)
- Purge (IF$PRG)
- Protect (IF$PRO)—directory files only
- Unprotect (IF$UNP)—directory files only

The appropriate request packets are sent to the ACP only when necessary to complete a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the ACP. Other Pascal statements unconditionally cause the OTS to issue send requests; examples are BREAK, which generates a Write Logical (IF$WTL),

and CLOSE, which generates a Close (IF$CLS) request (normally preceded by a Write Logical, unless BREAK immediately precedes CLOSE).

Pascal Get Characteristics functions that report the characteristics of disks are provided in the file GETSET.PAS on the MicroPower/Pascal distribution kit. Those functions issue Get Characteristics (IF$GET) request packets to the driver.

The following packet-level driver functions cannot be performed with normal Pascal I/O statements or GETSET functions:

- Read Physical (IF$RDP)
- Write Physical (IF$WTP)—except for RX02 formatting

To perform these functions, either use the request/reply packet interface directly or write Pascal procedures that take a user-specified file variable (or queue semaphore ID) and send the appropriate request packets to the driver. (The Get/Set Characteristics procedures in GETSET.PAS demonstrate the latter approach.)

4.4 Request/Reply Packet Interface

The packet-level functions provided by the disk-class device drivers are listed below by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical (RL01/2, RX02, TU58)
IF$RDL (1)	Read Logical
IF$WTP (3)	Write Physical (RL01/2, RX02, TU58)
IF$WTL (4)	Write Logical
IF$GET (7)	Get Characteristics
IF$ONY (8)	Bypass Only (MSCP— for internal use only)
IF$BYP (9)	Bypass (MSCP—for internal use only)
IF$INT (10)	Initialize Port (MSCP—for internal use only)

If a request is received for an Open (IF$LOK or IF$ENT), a Close (IF$CLS), or a Purge (IF$PRG), the driver returns an illegal function (ES$IFN), which the ACP interprets as indicating that no device-dependent processing was required for that operation.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the disk-class device drivers are shown below by symbolic code and bit position:

Code	Function
FM$WFM (bit 6)	Format device (RX02 Write Physical)
FM$WSD (bit 7)	Format single density (RX02 Write Physical)
FM$BSM (bit 13)	Signal binary/counting semaphore
FM$DCK (bit 14)	Data check (TU58)
FM$INH (bit 15)	Inhibit retries on error (RL01/2, RX02, MSCP, TU58)

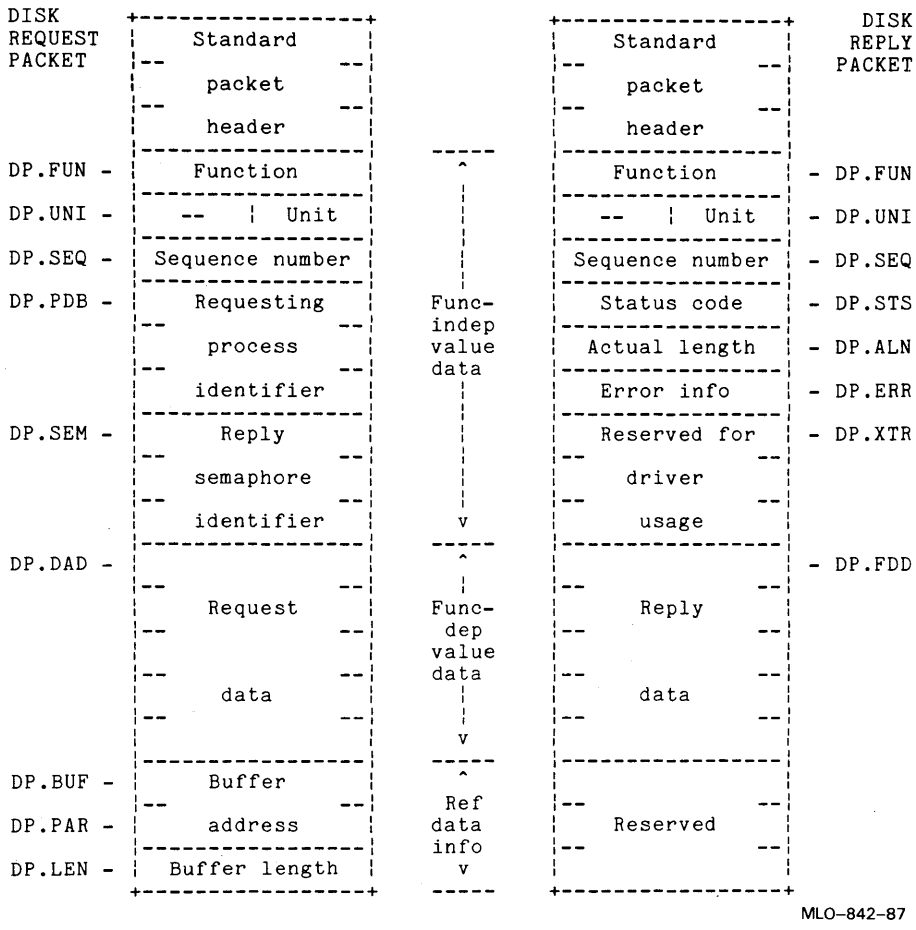
Each disk-class device driver consists of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each configured controller. (For the VM driver, "controller" means "memory region," as specified in the VM driver prefix file.) Multiple processes within a driver process family share the same instruction and pure-data segments but require separate RAM for impure data. I/O requests intended for a particular controller are sent (using a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The following shows request queue names and number of supported units for disk driver requests:

Driver	Request Queue Name	Number of Units	Numbering
RL01/2	$DLc	1–4 (any combination of RL01s and RL02s)	In prefix file
RX02	$DYc	1–2	0 for left drive and 1 for right in dual-drive
MSCP	$DUc	1–n	In prefix file
Extended disk	$XDc	1–n (partitions), as determined by physical disk size and user-defined partition size	0 through (n–1)
TU58	$DDc	1–2	0 for left drive and 1 for right in dual-drive
Virtual memory	$VMc	1	0

The letter c in a queue name represents a controller designation (A, B, ...—as specified in a driver prefix file). The number of units configured for each controller and their unit numbers must be specified in a disk driver prefix file. Typically, unit numbering starts at 0.

The general format of the disk request and reply packets follows:



MLO-842-87

The function-independent portions of the packets shown above are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of disk driver function.

Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

4.4.1 RL01/2 (DL) Functions

4.4.1.1 DL Logical Read and Write

An RL01/2 logical read or write operation transfers data to or from a user buffer, starting at a disk address that is specified in terms of a logical block number—0 to 10209 for the RL01, 0 to 20449 for the RL02.

The unit of storage implied by logical I/O operations is the 512-byte logical block, which consists of two logically contiguous sectors.

The disk driver converts logical block numbers into physical device addresses—tracks, cylinders, and sectors. Logical blocks span several sectors and may cross cylinders.

Multisector logical transfers read from or write to logically sequential sectors of the disk.

A write operation that does not fill the last or only block involved causes the remainder of the block to be zero-filled; this remainder can include the entire second sector of the block.

The RL01 disk has 20 logical blocks per track and 510.5 usable tracks, for a total of 10,210 logical blocks. The RL02 disk has 20 logical blocks per track and 1022.5 usable tracks, for a total of 20,450 logical blocks.

Note

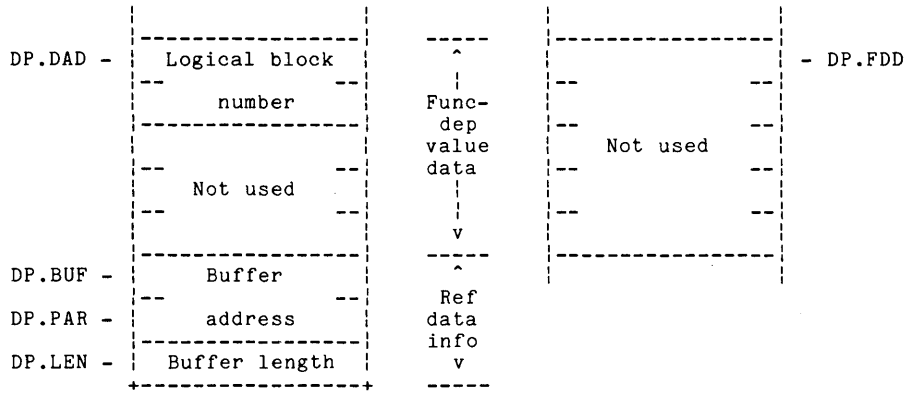
The last track on an RL01 or RL02 disk, containing the replacement blocks for bad-block replacement, is write-protected by the DL driver. This track is excluded from the calculation of usable logical blocks.

In addition, for RT–11 compatibility, the last 10 blocks on the next to last track of each disk are also excluded from the logical block calculation. The RT–11 RL01 and RL02 drivers reserve these 10 blocks for bad-block replacements. The DL driver does not use or write-protect these blocks but also does not include them in the device-dependent information it returns to the caller.

The format used for recording logical blocks is RT–11-compatible: twenty 2-sector logical blocks per track with a 34-sector per track offset.

All RL01/2 read and write operations transfer an even number of bytes to or from the user's buffer because of the word orientation of the device.

The following are function-dependent portions of the DL logical read or write request and reply packets:



The range of the logical-blk-num value is 0 to 10,209 for the RL01 or 0 to 20,449 for the RL02.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.1.2 DL Physical Read and Write

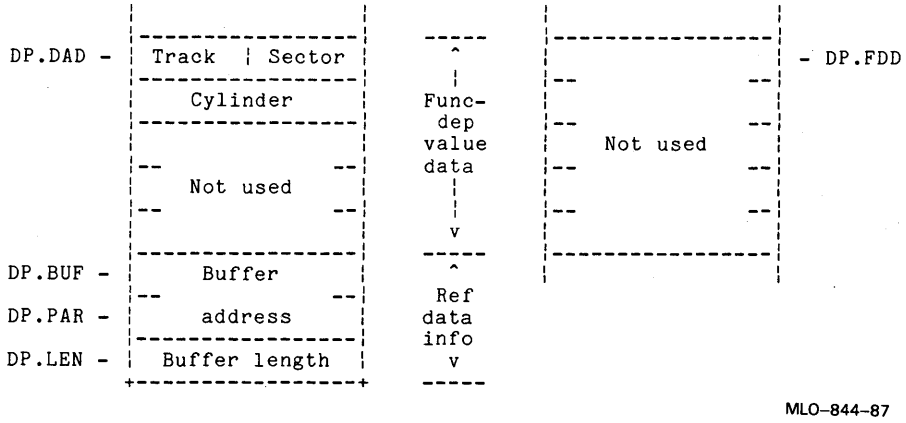
An RL01/2 physical read or write operation transfers data to or from the user's buffer, starting at a physical device address specified by absolute track, cylinder, and sector number.

The unit of storage implied by physical I/O operations is the 128-word sector. Data transfers can start at any physical sector of the disk.

A write operation that does not fill the last or only sector involved causes the remainder of the sector to be zero-filled.

All RL01/2 read and write operations transfer an even number of bytes to or from the user's buffer because of the word orientation of the device.

The following are function-dependent portions of the DL physical read or write request and reply packets:



MLO-844-87

The range of the sector value is 1 to 40.

The track value is 0 or 1.

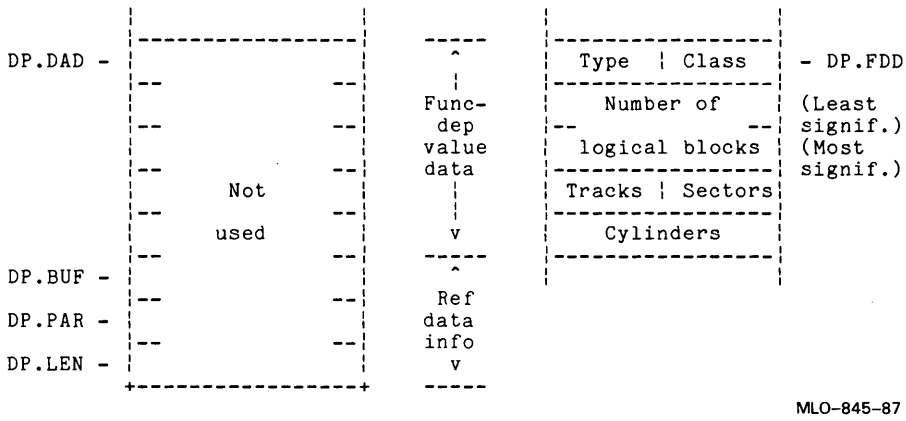
The range of the cylinder value is 0 to 255 for RL01 or 0 to 511 for RL02.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.1.3 DL Get Characteristics

The DL Get Characteristics function returns a block of device-dependent information about a specified RL01/2 unit in the function-dependent portion of the reply message. The information consists of the codes for device class and type, the number of logical blocks per unit—10,210 for the RL01 and 20,450 for the RL02—and the number of tracks (surfaces), sectors, and cylinders per unit. The unsafe volume (ES$UNS) error is returned if a disk is not properly mounted for a Get Characteristics request.

The following are function-dependent portions of the DL Get Characteristics request and reply packets:



MLO-845-87

In the reply information above:

- Class is DC$DSK for disk device class.
- Type is DK$DL for RL01/RL02 device type.
- The number of logical blocks, tracks, sectors, and cylinders is given per unit—for one disk. The number of tracks is reported as 2, indicating two recording surfaces.

4.4.2 RX02 (DY) Functions

4.4.2.1 DY Logical Read and Write

An RX02 logical read or write operation transfers data to or from a user buffer, starting at an initial disk address that is specified in terms of a logical block number—0 to 493 for single density, 0 to 987 for double density.

The unit of storage implied by logical I/O operations is the 512-byte logical block. In single-density mode, a logical block consists of four logically contiguous sectors; in double-density mode, two logically contiguous sectors. (The sectors are physically noncontiguous because of the 2:1 sector interleaving algorithm used to read and write logical blocks.)

The disk driver converts logical block numbers into physical device addresses—cylinders and sectors. Logical blocks span several sectors and may cross cylinders.

Multisector logical transfers read from or write to logically sequential sectors of the disk.

A write operation that does not fill the last or only block involved causes the remainder of the block to be zero-filled; this remainder can include the entire second sector of the block in double-density mode or as many as three complete sectors in single-density mode.

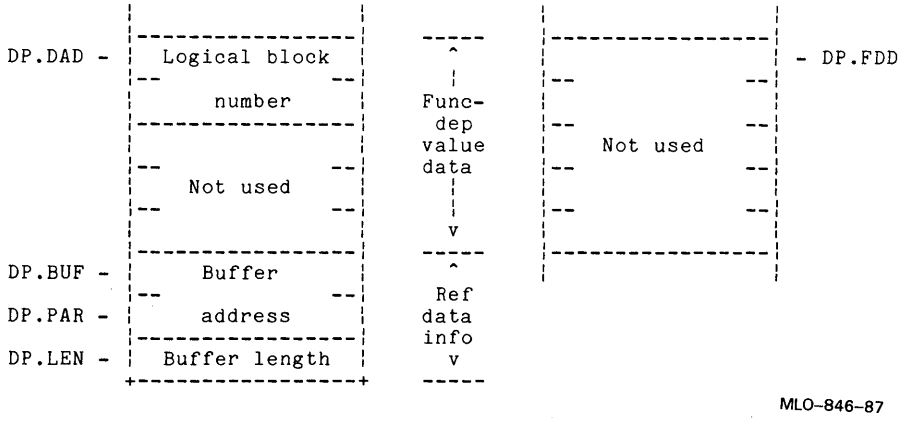
In accordance with DIGITAL and industry standards, cylinder 0 is unused in the organization of logical blocks on an RX02 diskette; logical block 0 begins at cylinder 1, sector 1. A single-density diskette has 6.5 logical blocks per cylinder and 76 usable cylinders, for a total of 494 logical blocks. A double-density diskette has 13 blocks per cylinder and 76 usable cylinders, for a

total of 988 logical blocks. (The logical block-recording technique used is RT–11-compatible: 2:1 interleaving with a 6-sector per cylinder offset.)

All RX02 read or write operations transfer an even number of bytes to or from the user's buffer because of the word orientation of the device. If an odd-value buffer length is specified in the request (field DP.LEN), the driver assumes one byte as the effective transfer length.

All read or write operations are tried at the density of the last request; the first request is always tried at single density. If a density error occurs and if retries are inhibited, the opposite density is set, and the ES$IVM status code is returned to the application program. (The user's program may then retry the previous request at the new density, if desired; in any case, the new density will be in effect for the next I/O operation performed on the drive unit.) If a density error occurs and if retries are not inhibited, the opposite density is set, and the request is retried automatically. If the density error persists after 10 retries, the ES$IVM status code is returned to the application program.

The function-dependent portions of the DY logical read or write request and reply packets are shown below:



The range of the logical-blk-num value is 0 to 493 for a single-density RX02 or 0 to 987 for a double-density RX02.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.2.2 DY Physical Read and Write

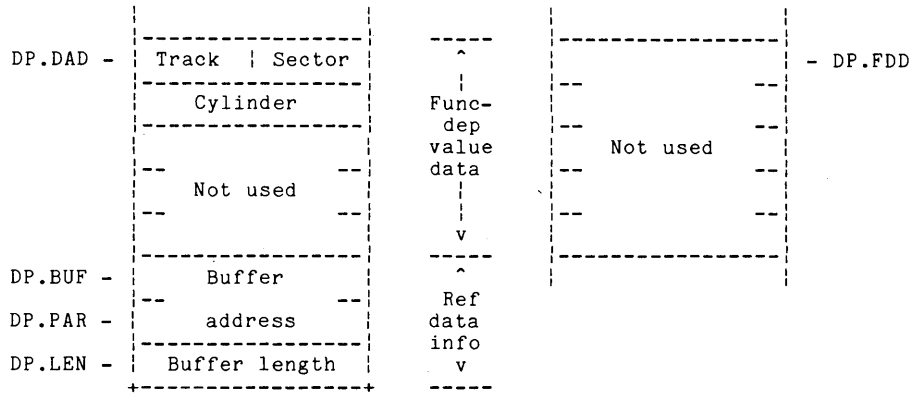
An RX02 physical read or write operation transfers data to or from the user's buffer, starting at a physical device address specified by absolute cylinder and sector number.

The unit of storage implied by physical I/O operations is the 64-word (single-density) or 128-word (double-density) sector. Data transfers can start at any physical sector of the diskette.

A write operation that does not fill the last or only sector involved causes the remainder of the sector to be zero-filled.

Two special forms of the physical write function format an RX02 diskette for single-density or double-density operation. (See the section on DY format subfunctions.)

The following are function-dependent portions of the DY physical read or write request and reply packets:



MLO-847-87

The range of the sector value is 1 to 26.

The track value is 0.

The range of the cylinder value is 0 to 76.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.2.3 DY Format Subfunctions of Physical Write

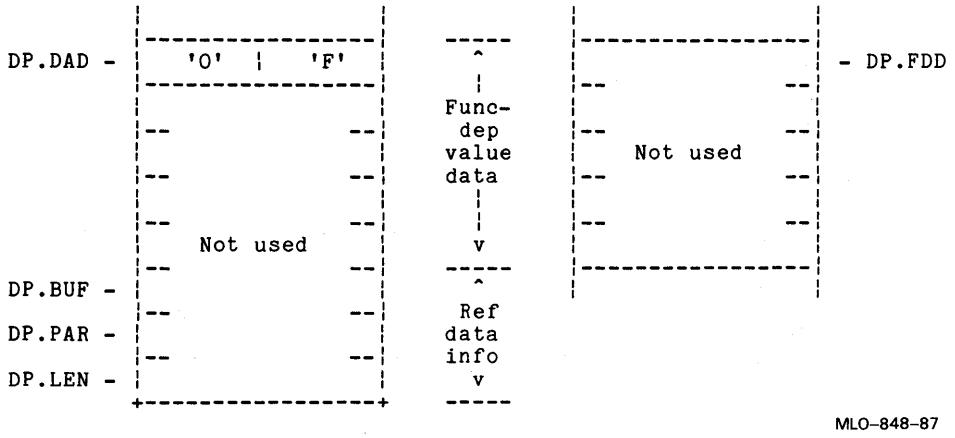
If modifier bits FM$WFM and FM$WSD of the function word are both set in an RX02 Write Physical (IF$WTP) function request, the meaning of the function is "format diskette for single-density;" if modifier bit FM$WFM is set and modifier bit FM$WSD is not set, the meaning of the function is "format diskette for double-density."

The single-density format subfunction reformats a double-density or single-density diskette for single density, clearing the entire volume in the process. The double-density format subfunction reformats a single-density or double-density diskette for double density, likewise clearing the entire volume.

Note

A format operation requires approximately 30 seconds to complete.

The function-dependent portions of the request and reply packets for the single- and double-density formatting subfunctions of Write Physical are shown below:



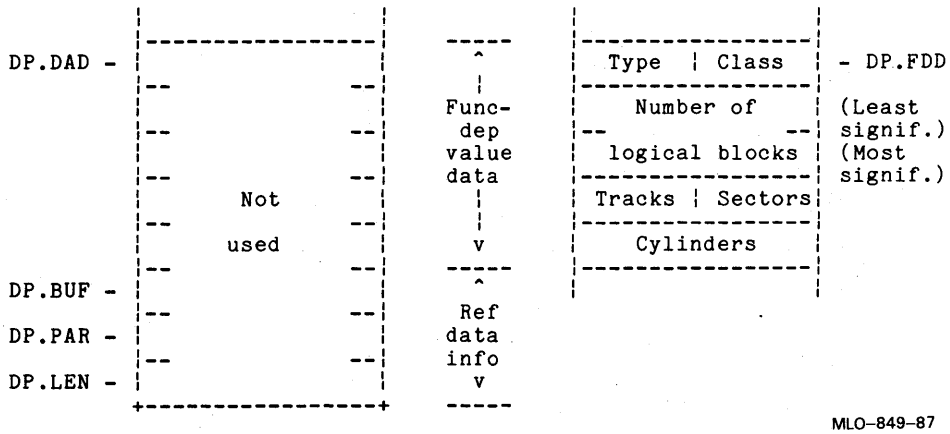
MLO-848-87

The DP.DAD field must contain the ASCII character sequence FO in the first (low-order) word.

4.4.2.4 DY Get Characteristics

The DY Get Characteristics function returns a block of device-dependent information about a specified RX02 unit in the function-dependent portion of the reply message. The information consists of codes for the device class and type, the number of logical blocks per unit—494 for single density, 988 for double density—and the number of tracks (surfaces), sectors, and cylinders per unit. The unsafe volume (ES$UNS) error is returned if a disk is not properly mounted for a Get Characteristics request.

The following are function-dependent portions of the DY Get Characteristics request and reply packets:



MLO-849-87

In the preceding reply information:

- Class is DC$DSK for disk device class.
- Type is DK$DY2 for RX02 device type.
- The number of logical blocks, tracks, sectors, and cylinders is given per unit—for one diskette. The number of tracks is reported as 1, indicating a single recording surface.

4.4.3 MSCP (DU) Functions

4.4.3.1 DU Logical Read and Write

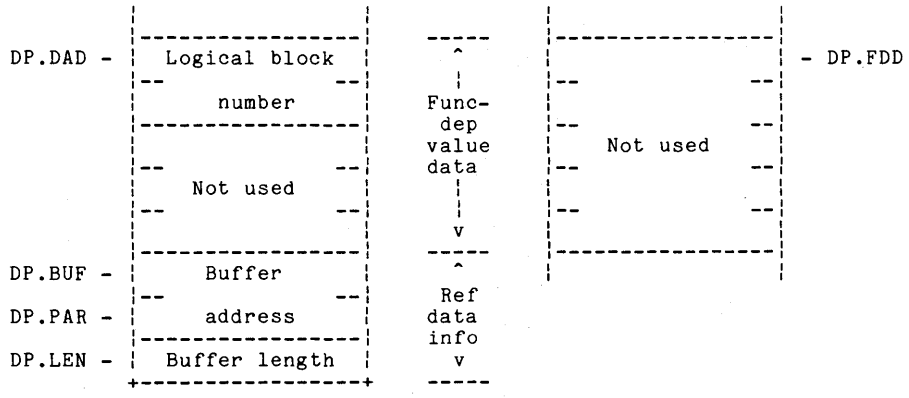
An MSCP logical read or write operation transfers data to or from the user's buffer, starting at a 512-byte logical block specified by a logical block number—0 to n–1, where n is the size of the disk in logical blocks.

The unit of storage implied by logical I/O operations is the 512-byte logical block.

A write operation that does not fill the last or only block involved causes the remainder of the block to be zero-filled.

Read and write operations to an MSCP disk transfer an even number of bytes to or from the user's buffer because of the word orientation of the devices.

The following are function-dependent portions of the DU logical read or write request and reply packets:



MLO-850-87

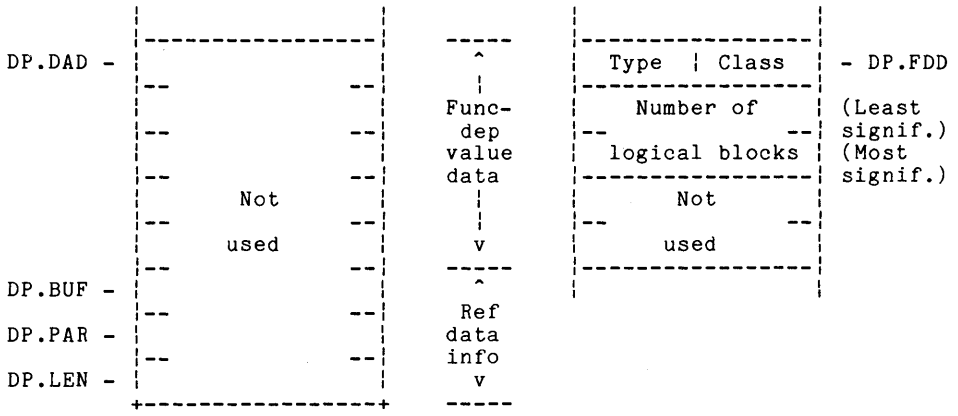
The range of the logical-block-number value is 0 to n–1, where n is the size of the device in logical blocks.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.3.2 DU Get Characteristics

The DU Get Characteristics function returns a block of device-dependent information about a specified MSCP unit in the function-dependent portion of the reply message. The information consists of the codes for device class and type and the number of logical blocks per unit. The only way to distinguish between MSCP disks is by the number of logical blocks per unit. The unsafe volume (ES$UNS) error is returned if a disk is not properly mounted for a Get Characteristics request.

The following are function-dependent portions of the DU Get Characteristics request and reply packets:



MLO-851-87

In the reply information above:

- Class is DC$DSK for disk device class.
- Type is DK$DU for MSCP disk device type.
- The number of logical blocks is given per unit—for one disk.

4.4.4 Extended Disk (XD) Functions

4.4.4.1 XD Logical Read and Write

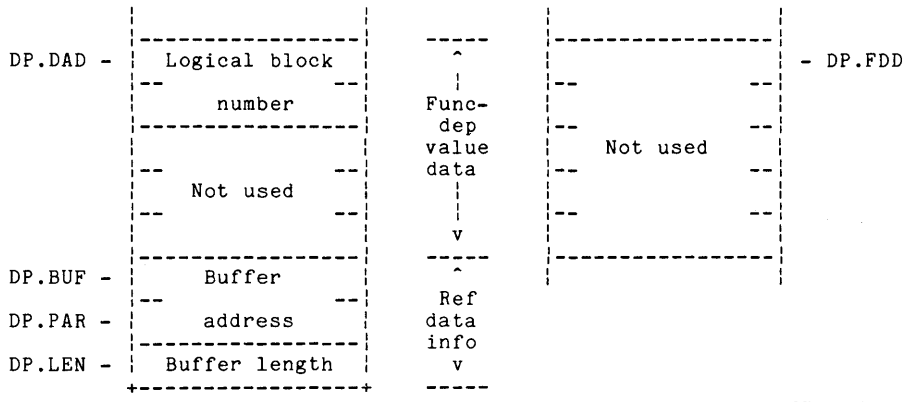
The XD packet-level read/write functions normally are accessed through the file system, not through explicit send requests from the user process. You can issue send requests for extended disk I/O to the XD driver, although normally it is preferable to issue the requests directly to the physical disk device driver. (An exception would be if the XD driver is present in your application and is accessed both by the file system and by sends, by different processes.)

An extended disk logical read or write operation transfers data to or from the user's buffer, starting at a 512-byte logical block specified by a logical block number—0 to n–1, where n is the size of the partition in logical blocks.

The unit of storage implied by logical I/O operations is the 512-byte logical block.

See the descriptions of the MicroPower/Pascal physical disk device drivers for information on zero filling of blocks, word or byte orientation of devices, and so forth. An XD driver transfer takes on the characteristics of the physical-disk driver on which XD is layered.

The following are function-dependent portions of the XD logical read or write request and reply packets:



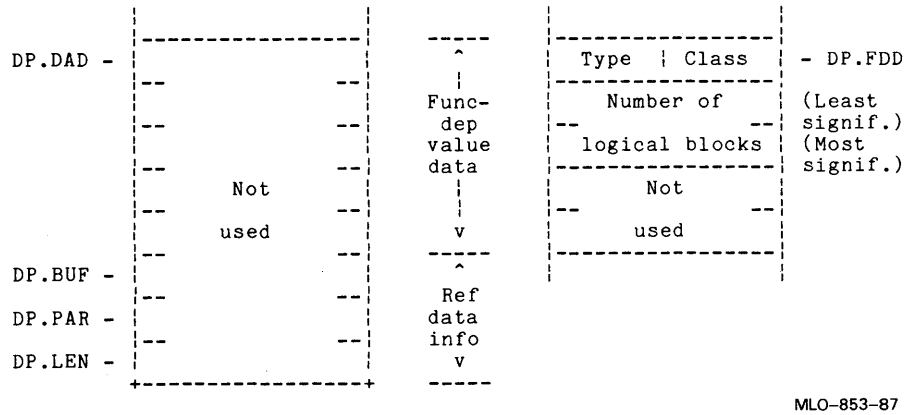
MLO-852-87

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.4.2 XD Get Characteristics

The XD Get Characteristics function returns a block of device-dependent information about a specified XD partition in the function-dependent portion of the reply message. The information consists of the codes for device class and type and the number of logical blocks in the partition.

The following are function-dependent portions of the XD Get Characteristics request and reply packets:



MLO-853-87

In the reply information above:

- Class is DC$DSK for disk device class.
- Type is DK$XD for extended disk device type.
- The number of logical blocks is given for the requested partition (DP.UNI).

4.4.5 TU58 (DD) Functions

4.4.5.1 DD Logical Read and Write

A TU58 logical read or write operation transfers data to or from a user buffer, starting at a tape address that is specified in terms of a logical block number (0 to 511).

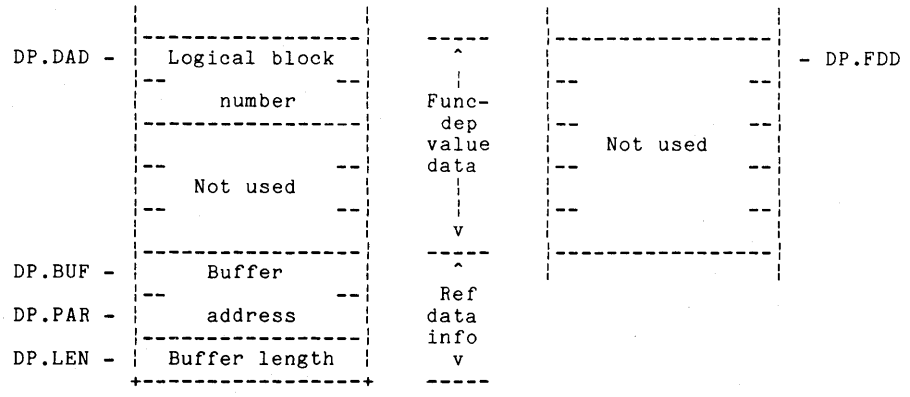
The unit of storage implied by logical I/O operations is the 512-byte logical block, which consists of four logically contiguous 128-byte records. (The records are physically noncontiguous because of the automatic interleaving performed by the controller in normal tape addressing mode.)

A write operation that does not fill the last or only block involved causes the remainder of the block to be zero-filled; that remainder can consist of up to 511 bytes.

If modifier bit FM$DCK of DP.FUN is set to 1 in a TU58 read function request, the driver instructs the drive to read with increased threshold. That type of read operation can be used to check for fading data on the tape.

If modifier bit FM$DCK of DP.FUN is set to 1 in a TU58 write function request, the driver instructs the drive to write with read verify. Following the write portion of the request, the drive attempts to read the data without errors; the drive returns a status code to the driver, indicating success or failure. That type of write operation can be used to ensure that reliable data can later be recovered.

The following are function-dependent portions of the DD logical read or write request and reply packets:



MLO-854-87

The range of the logical-blk-num value is 0 to 511.

The buffer-length value determines the length, in bytes, of the data transfer.

If retries were required to complete the operation successfully, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply packet, and a value of 1 is returned in the error-info (DP.ERR) field of the packet. The status-code and error-info fields are in the function-independent portion of the packet.

4.4.5.2 DD Physical Read and Write

A TU58 physical read or write operation transfers data to or from the user's buffer, starting at a device address specified by a physical record number (0 to 2047). (Tape positioning is performed in special address mode.)

The unit of storage implied by physical I/O operations is the 128-byte record. Data transfers can start at any physical record on the tape.

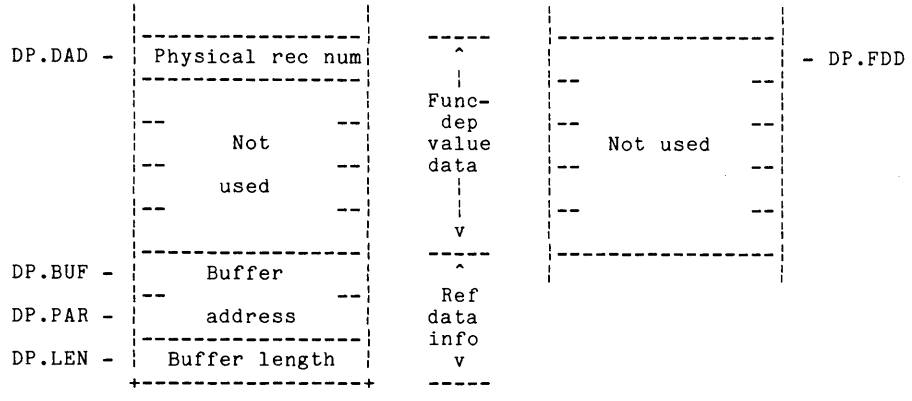
Multirecord transfers—exceeding 128 bytes—read from or write to physically sequential records on the tape.

A write operation that does not fill the last or only record involved causes the last record (up to 127 bytes) to be zero-filled. The standard record interleaving that is performed for logical I/O is disabled for physical I/O.

If modifier bit FM$DCK of DP.FUN is set to 1 in a TU58 read function request, the driver instructs the drive to read with increased threshold. That type of read operation can be used to check for fading data on the tape.

If modifier bit FM$DCK of DP.FUN is set to 1 in a TU58 write function request, the driver instructs the drive to write with read verify. Following the write portion of the request, the drive attempts to read the data without errors; the drive returns a status code to the driver, indicating success or failure. That type of write operation can be used to ensure that reliable data can later be recovered.

The following are function-dependent portions of the DD physical read or write request and reply packets:



MLO-855-87

The range of the physical-rec-num value is 0 to 2047.

The buffer-length value determines the length, in bytes, of the data transfer.

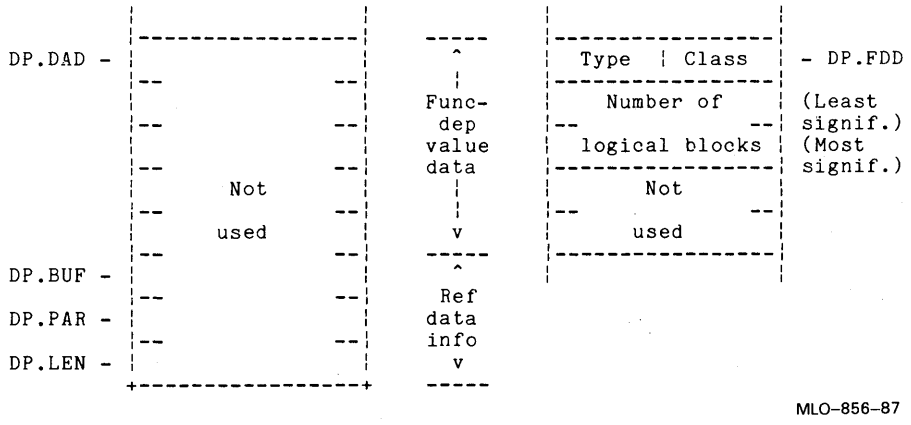
If retries were required to complete the operation successfully, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply packet, and a value of 1 is returned in the error-info (DP.ERR) field of the packet. The status-code and error-info fields are in the function-independent portion of the packet.

4.4.5.3 DD Get Characteristics

The DD Get Characteristics function returns a block of device-dependent information about a specified TU58 unit in the function-dependent portion of the reply message. The information consists of the codes for device class and type and the number of logical blocks per unit.

In addition to returning device characteristics, the Get Characteristics function performs a seek operation—to the first block of the directory—in order to determine whether a cartridge is in the drive. If no cartridge is in the drive, the unsafe volume (ES$UNS) status code is returned in the function-independent portion of the reply message.

The following are function-dependent portions of the DD Get Characteristics request and reply packets:



MLO-856-87

In the reply information above:

- Class is DC$DSK for disk device class.
- Type is DK$DD (TU58) for TU58 device type.
- The number of logical blocks is given per unit—for one DECtape II cartridge (always 512 blocks).

4.4.6 Virtual Memory (VM) Functions

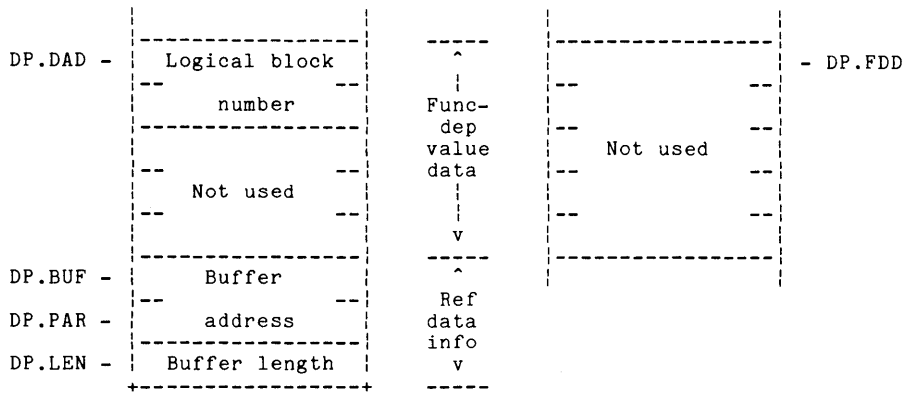
4.4.6.1 VM Logical Read and Write

A virtual-memory-disk logical read or write operation transfers data to or from the user's buffer, starting at a 512-byte logical block specified by a logical block number—0 to n–1, where n is the size of the memory region in logical blocks.

You specify the memory region size in the VM driver prefix file. The unit of storage implied by logical I/O operations is the 512-byte logical block, although of course the number of blocks you specify is converted to 64-byte PAR ticks for the purpose of allocating the region.

When a read or write request is received, the driver maps the "device" and the user buffer to PARs 0 and 1 and then uses the driver subroutine $BLXIO to copy the logical block. ($BLXIO is described in Chapter 15.) An even number of bytes are transferred between the user buffer and virtual memory using word move (MOV) instructions. A write operation that does not fill the last or only block involved causes the remainder of the block to be zero-filled.

The following are function-dependent portions of the VM logical read or write request and reply packets:



MLO-857-87

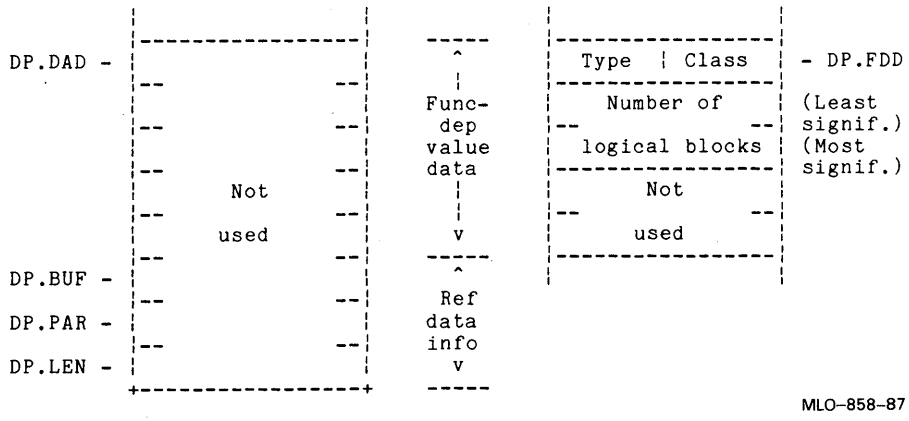
The range of the logical-blk-num value is 0 to n–1, where n is the size of the device in logical blocks.

The buffer-length value determines the length, in bytes, of the data transfer.

4.4.6.2 VM Get Characteristics

The VM Get Characteristics function returns a block of device-dependent information about a specified virtual memory "disk" unit in the function-dependent portion of the reply message. The information consists of the codes for device class and type and the number of logical blocks in the memory region (as specified in the VM driver prefix file).

The function-dependent portions of the VM Get Characteristics request and reply packets are shown below:



MLO-858-87

In the preceding reply information:

- Class is DC$DSK for disk device class.
- Type is DK$VM for virtual memory disk device type.

4.5 Status Codes

If an error is detected during an I/O operation by a disk-class device or driver, the driver returns an exception code in the status-code field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error was detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply message.

The disk drivers return the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	Driver process deleted; request not serviced
ES$CTL	HARD_IO	Controller error (DY, DU)
ES$DRV	HARD_IO	Drive error (DL, DU); all retries failed, data check error, seek error (block not found), motor stopped, or bad operation code (DD)
ES$FOR	HARD_IO	Media format error (DU)
ES$IBN	HARD_IO	Invalid block number on read/write request (DU, DD, XD)
ES$IDA	HARD_IO	Invalid device address on read/write request (DL, DY, VM)
ES$IVD	HARD_IO	Invalid data (DU)
ES$IVM	HARD_IO	Invalid mode—volume formatted for opposite or unrecognized density (DY)
ES$IVP	HARD_IO	Invalid request packet parameter—odd buffer address or odd number of bytes to transfer (DL, DY, DU, VM)
ES$NXM	HARD_IO	Attempted transfer to nonexistent memory or write to ROM (DL, DY)
ES$NXU	HARD_IO	Nonexistent unit—unit number not defined in prefix file (DL, DU); drive number greater than 1 (DY, DD); unit number greater than 0 (VM); unit number not defined in XDDRV (XD)
ES$OFL	HARD_IO	Unit off line (DU)
ES$OVF	HARD_IO	Data overflow (DY)
ES$PAR	HARD_IO	Parity error, CRC error (DL); unrecoverable CRC error or soft error with no retry (DY)
ES$PWR	HARD_IO	Device power failure (DY)
ES$UNS	HARD_IO	Unsafe volume, drive not ready: door open, power not OK, drive not up to speed, no volume, no cartridge in drive (DL, DY, DU, DD)
ES$WLK	HARD_IO	Write-locked unit (DL, DU, DD)
ES$IFN	SOFT_IO	Illegal function code
ES$IVL	SOFT_IO	Invalid length specified (XD)
ES$NRF	SOFT_IO	No reference data present (DD)

Exception codes are defined in the EXC.PAS include file for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for OTS-detected I/O errors or for kernel-detected errors that the disk drivers raise rather than passing back to the requesting process. OTS-detected I/O errors are listed in Chapter 9 of the *MicroPower/Pascal Language Guide.*

4.6 Extended Error Information

The RL01/2 (DL), RX02 (DY), and TU58 (DD) disk-class drivers return extended error information to packet-level users.

In the event of a hardware error, the DL driver copies the multipurpose register (MPR) into the DP.ERR field of the reply packet. See the *RLV12 Disk Controller User's Guide* for a description of the MPR.

In the event of a hardware error, the DY driver copies one byte of definitive error code—as returned by the RXV21 in response to the read error code function—into the DP.ERR field of the reply packet. That status information is described in the *RX02 Floppy Disk System User's Guide.*

For all status returns and hardware error returns in particular, the DD driver returns a hardware success code in the low-order byte of the DP.ERR field of the reply message. (It is the same hardware success code returned by the TU58 controller for each operation in byte 3 of the end packet.) In the event of a hardware error, the hardware success code provides more specific error information. See the *TU58 DECtape II User's Guide* for a description of the end packet sent by the tape controller.

4.7 Disk Driver Prefix Files

Figures 4–1 through 4–5 show the disk driver prefix modules. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

Note

No prefix module exists for the XD driver. Instead, you edit the XD driver source module, available on the distribution kit, to fit your application. See Section 4.8 for details.

The DRVCF$ macro contains a field (nctrl) for the number of controllers (or memory regions) on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request-queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced by the driver. It gives the controller name, number of units, CSR and vector addresses, and unit numbers. You can edit those fields, if your controller does not conform to the defaults. For the VM driver, the memory region size—the number of 512-byte blocks—is specified in place of the CSR and vector addresses. For the DD driver, the serial line type and speed are specified. The five CTRCF$ invocations in the DD driver prefix file specify a DLV11-type SLU; a KXT11–CA, FALCON, or CMR21 console DLART; a KXJ11–CA console DLART; an SBC–11/21 DLART type SLU; and a KXT11–CA or KXJ11–CA multiprotocol channel B (SLU2B), each with a line speed of 38400. (See Chapter 3 for valid serial line types and speeds.)

Note

The DU prefix file shows two possible CTRCF$ definitions for a single controller rather than CTRCF$ definitions for multiple controllers (the normal practice).

The units field specifies the unit numbers of the devices attached to the controller. The designation 0:1 refers to unit 0 and unit 1. For an RX02 or a TU58, 0 and 1 are the only possible unit numbers, but you can edit that field if you have only one unit <0> . Note that <0,1> and <0:1> are equivalent. For a virtual memory region, 0 is the only possible unit number.

The interrupt vectors specified in those macros must also be specified in the system configuration file, using the DEVICES macro.

The xxIPR, xxPPR, and xx$HPR definitions specify the initialization and request-handling software priorities for the disk driver processes and the hardware interrupt priority for the disk controllers. All controllers associated with a given driver have the same priority. Of course, no hardware interrupt priority is specified for a virtual memory "controller" (region).

Figure 4-1: RL01/RL02 Driver Prefix File (DLPFX.MAC)

```
   .TITLE   DLPFX   - RLV11, RLV21 Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1982, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
   .mcall   drvcf$, ctrcf$

DL$IPR   ==   250.       ; Process initialization priority
DL$PPR   ==   175.       ; Process priority
DL$HPR   ==   4          ; RLV11 hardware priority

   drvcf$   dname=DL,nctrl=1
   ctrcf$   cname=A,nunits=2.,csrvec=<174400,160>,units=<0:1>
;  ctrcf$   cname=B,nunits=2.,csrvec=<174410,164>,units=<0,1>

   .end
```

Figure 4–2: RX02 Driver Prefix File (DYPFX.MAC)

```
   .TITLE   DYPFX   - RX02 Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1982, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
   .mcall   drvcf$, ctrcf$

DY$IPR   ==   250.      ; Process initialization priority
DY$PPR   ==   175.      ; Process priority
DY$HPR   ==   5         ; RX02 hardware priority

   drvcf$   dname=DY,nctrl=1
   ctrcf$   cname=A,nunits=2.,csrvec=<177170,264>,units=<0:1>
;  ctrcf$   cname=B,nunits=2.,csrvec=<177200,270>,units=<0,1>

   .end
```

Figure 4–3: MSCP Disk-Class Driver Prefix File (DUPFX.MAC)

```
   .TITLE   DUPFX   - MSCP Micro PDP-11 Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1983, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
   .mcall   drvcf$, ctrcf$

DU$IPR   ==   250.      ; Process initialization priority
DU$PPR   ==   175.      ; Process priority
    drvcf$   dname=DU,nctrl=1
    ctrcf$   cname=A,nunits=3.,csrvec=<172150,154>,units=<0:2>
;   ctrcf$   cname=A,nunits=3.,csrvec=<174344,154>,units=<0:2>

    .end
```

Figure 4-4: TU58 Driver Prefix File (DDPFX.MAC)

```
.nlist
   .enabl   LC
.list
   .TITLE   DDPFX ¦ - TU58 Device Driver Prefix Module
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL
;  RIGHTS RESERVED.
;
   .mcall   drvcf$
   .mcall   ctrcf$
   .mcall   drvdf$
   drvdf$
DD$PPR   ==   175.       ; Process priority
DD$HPR   ==   4          ; TU58 hardware priority (must be 5 for Falcon
                         ; SLU2)
DD$IPR   ==   250.       ; Process initialization priority
   drvcf$   dname=DD,nctrl=1
;DLV11 type SLU
;   ctrcf$   cname=A,nunits=2.,csrvec=<176510,310>,units=<0:1>,typrm=<TT$DL,
;38400>
;KXT11--CA/FALCON/CMR21 Console DLART
;   ctrcf$   cname=A,nunits=2.,csrvec=<176560,60>,units=<0:1>,typrm=<TT$DLT,
;38400>
;KXJ11--CA Console DLART
;   ctrcf$   cname=A,nunits=2.,csrvec=<176560,60>,units=<0:1>,typrm=<TT$DLU,
;38400>
;FALCON SLU2 DLART
; * Remember to change DD$HPR to 5 if Falcon SLU2 DLART is selected *
;   ctrcf$   cname=A,nunits=2.,csrvec=<176540,120>,units=<0:1>,typrm=<TT$DLT,
;38400>
;KXT11--CA or KXJ11--CA Multiprotocol channel B (SLU2B)
;   ctrcf$   cname=A,nunits=2.,csrvec=<175710,160>,units=<0:1>,typrm=<TT$DM,
;38400>
   .end
```

Figure 4–5: Virtual Memory Driver Prefix File (VMPFX.MAC)

```
   .TITLE   VMPFX   - Virtual Memory Driver Prefix Module
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
   .mcall   drvcf$, ctrcf$

VM$IPR   ==   250.      ; Process initialization priority
VM$PPR   ==   175.      ; Process priority

VMSIZ    =    <128.>    ; size in blocks (each block is 512. bytes)

   drvcf$   dname=VM,nctrl=1
   ctrcf$   cname=A,nunits=1.,csrvec=<VMSIZ>,units=<0>

   .end
```

4.8 Extended Disk Driver Source Excerpt

Figure 4–6 shows the portions of the extended disk (XD) driver source, XDDRV.PAS, that can be edited to fit your application.

MAX_UNIT_SIZE, the maximum number of disk blocks for each partition, must not exceed 65,536. If there is room for multiple partitions on the disk, every partition, except possibly the last, will be of this size.

MIN_UNIT_SIZE, the minimum number of disk blocks for each partition, gives the minimum acceptable value for the last (or only) partition on the disk.

To illustrate the use of MAX_UNIT_SIZE and MIN_UNIT_SIZE, an extended disk with 138,000 blocks for which you specify MAX_UNIT_SIZE = 65000 and MIN_UNIT_SIZE = 100 will have two 65,000 block partitions (XDA0: and XDA1:) and one 8000 block partition (XDA2:).

NO_DEVICES specifies the number of physical disks to be partitioned.

The PARTITION_ONE_PHYSICAL_UNIT procedure must be called NO_DEVICES times. For each physical disk to be partitioned, PARTITION_ONE_PHYSICAL_UNIT identifies the request queue semaphore name and unit number associated with the physical disk (DEVICE_NAME_X and UNIT_NUMBER_X) and the XD request queue semaphore name to be associated with that disk unit (XD_NAME_X). INDEX indexes into the internal array that holds the control information; by convention, the INDEX value begins at 1 and increments up to NO_DEVICES.

For each extended disk after the first, increase the DATA_SPACE attribute by 456.

Figure 4–6: Extended Disk Driver Source File (XDDRV.PAS) Excerpt

```
[system(micropower), init_priority(250), priority(175),
      privileged, data_space(1000)]
   program $xddrv;

(*
                  COPYRIGHT (c) 1986 BY
             DIGITAL EQUIPMENT CORPORATION, MAYNARD
              MASSACHUSETTS.  ALL RIGHTS RESERVED.
*)

{
The XD driver permits MicroPower/Pascal to do perform file-structured
I/O to disk devices with
greater than 65536 blocks. It does this by partitioning the
physical disk unit into multiple partitions. Each physical disk
unit is treated as a single controller with one or more logical units.
}

 %include 'src:exc'
 %include 'src:iopkts'
 %include 'src:gsinc'

  const

{Define the maximum number of disk blocks per partition.}
{NOTE: Must not exceed 65536.}
    max_unit_size = 65536;
{Define the minimum number of disk blocks per partition.}
    min_unit_size = 100;
{Define the number of physical disk units which are to be partitioned.}
    no_devices = 1;
    .
    .
    .
{Procedure partition_one_physical_unit is invoked no_devices times.}

   partition_one_physical_unit(
   device_name_x := '$DUA  ',
   unit_number_x := 0,
   xd_name_x := '$XDA  ',
   index := 1);
{
Note: For each additional call to partition_one_physical_unit,
      increase data_space attribute by 456 (dynamic process
      stack and impure area).

   partition_one_physical_unit(
Request queue semaphore name associated with physical disk device.
   device_name_x := '$DUA  ',
Unit number of physical disk device.
   unit_number_x := 1,

Request queue semaphore name associated with extended disk.
Should be of the form '$XDc  '.
   xd_name_x := '$XDB  ',
Ordinal in range 1,2,...,no_devices.
   index := 2);
}
    .
    .
    .
  end. {$xddrv}
```

Chapter 5
TMSCP Tape Driver

This chapter describes the use of the MicroPower/Pascal TMSCP magnetic tape (MU) driver. The MU driver supports nondirectory-structured I/O operations on tape interfaces that use the Tape Mass Storage Control Protocol (TMSCP)—in particular, the TK50 streaming cartridge tape interface. The TK50 is used primarily for large-volume data storage or redundant (backup) storage by MicroPower/Pascal applications.

Note

TMSCP is a high-level interface to a family of tape controllers and devices manufactured by DIGITAL.

5.1 MU Driver Features and Capabilities

The MU driver supports read and write operations and the returning of device characteristics, plus the tape-specific operations Reposition, Write Tape Mark, and Rewind.

Read and write operations transfer data to or from a buffer in the calling process, starting at the current tape position.

The Get Characteristics operation reports the TMSCP device class and type.

The Reposition Tape operation repositions the tape to an offset forward or backward from the current position or forward from the beginning of tape (BOT), as determined by user-specified modifiers.

The Write Tape Mark operation establishes the end of a logical file.

The Rewind Tape operation repositions to the BOT.

5.2 Performing TMSCP Tape I/O

For most MicroPower/Pascal applications and particularly for streaming applications, you perform TMSCP tape I/O by invoking Pascal support routines—READ_TAPE, REWIND_TAPE, and so forth. Those routines provide high-level nonfile access to TMSCP tape controllers. The MU support routines issue Pascal send requests to the request queue semaphore of the MU driver. The routines are described in Section 5.3.

You can also perform TMSCP tape I/O by invoking Pascal I/O procedures that open files for tape data and then input or output the data in accordance with the rules for Pascal I/O. The Pascal I/O procedures—OPEN, GET, WRITE, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide*. However, file-oriented operations are of limited use; one limitation is that MU tape-specific operations cannot be performed by Pascal I/O procedures. Optionally, you can modify the Pascal support routines that perform Reposition, Write Tape Mark, and Rewind operations to accept a file variable. (For a model, see the Get and Set Characteristics functions in kit file GETSET.PAS.) A more serious limitation is that because of the nature of Pascal buffering, you must take special care to provide the necessary degree of input or output synchronization. Pascal I/O is unsuitable for streaming applications.

In addition to invoking the TMSCP support routines or Pascal I/O procedures, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the TMSCP controller interrupt vector addresses
2. Edit the MU driver prefix file to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, number of controller units (1) and identifying number (0)
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - MU driver process
   - Pascal TMSCP support routines (from kit files MUSUB.PAS and MUINC.PAS)
   - [For MU file OPEN only:] Ancillary control process (ACP)
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any other I/O support routines you opt to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for TMSCP tape I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 5.7 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

Alternatives to using the TMSCP support routines or the Pascal I/O procedures for TMSCP tape I/O exist, but require more effort. You can:

- Issue your own Pascal or MACRO–11 packet-level requests to the driver, bypassing the Pascal support routines for nonfile I/O, as well as the ACP and the OTS file routines (low-level nonfile access)
- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and the driver, bypassing the OTS file routines (lower-level file system access)

The following sections describe the Pascal support routine interface to the MU driver, the Pascal I/O procedure interface, the lower-level request/reply packet interface, the status codes that can be returned to users of any interface, and the MU driver prefix file.

5.3 Pascal Support Routine Interface

The following support routines, written in Pascal and independent of the file system, provide a high-level interface to TMSCP tape controllers:

- READ_TAPE procedure
- WRITE_TAPE procedure
- REPOSITION_TAPE procedure
- WRITE_TAPE_MARK procedure
- REWIND_TAPE procedure
- INIT_TAPE_CNTL procedure—for internal use only

Note

The TMSCP support routines use all of the packet-level MU driver functions except Get Characteristics (IF$GET). To perform that operation, use the Get Characteristics function (descriptor version) in the distribution kit file GETSET.PAS.

The following sections describe Pascal support routines for TMSCP tape I/O. Each routine allocates an I/O packet, fills it in with information based on the procedure parameters, sends it to the MU driver queue semaphore for the specified port, and returns immediately to the caller. If the routine has a reply parameter, the driver sends a standard driver reply via the specified queue semaphore when the operation is complete. (The driver reply packets are described in Section 5.5.)

Note

The distributed support routines assume a unit number of 0 for each operation.

The following files on the MicroPower/Pascal distribution kit are required for using the routines:

File	Description
MUSUB.PAS	TMSCP routine source module
MUINC.PAS	TMSCP routine include file
IOPKTS.PAS	Pascal I/O include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

5.3.1 READ_TAPE

The READ_TAPE procedure requests a read operation, which transfers data to the user's buffer from the current tape position. The length of the specified buffer determines the length of the data transfer.

The packet-level equivalent of READ_TAPE is the IF$RDL function.

The syntax for calling the procedure is as follows:

```
READ_TAPE ( buffer, mu_desc, reply );
```

Parameter	Type	Description
VAR buffer	PACKED ARRAY [first..last: INTEGER] of CHAR	Data buffer
VAR mu_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The count of bytes transferred is returned in the actual-length field of the MU driver reply packet.

5.3.2 WRITE_TAPE

The WRITE_TAPE procedure requests a write operation, which transfers data from the user's buffer to the current tape position. The length of the specified buffer determines the length of the data transfer.

The packet-level equivalent of WRITE_TAPE is the IF$WTL function.

The syntax for calling the procedure is as follows:

```
WRITE_TAPE ( buffer, mu_desc, reply );
```

Parameter	Type	Description
VAR buffer	PACKED ARRAY [first..last: INTEGER] of CHAR	Data buffer
VAR mu_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The count of bytes transferred is returned in the actual-length field of the MU driver reply packet.

5.3.3 REPOSITION_TAPE

The REPOSITION_TAPE procedure requests that the tape be repositioned to a point that is specified either by use of a generic object count or by use of record and tape-mark counts.

Note

For tape-specific operations, the relevant units are records, tape marks (which indicate the end of a logical file), and objects (a context-dependent term for either records or tape marks).

Depending on user-specified modifiers, the tape can be repositioned to an offset forward or backward from the current position or forward from the BOT.

The packet-level equivalent of REPOSITION_TAPE is the IF$REP function.

The syntax for calling the procedure is as follows:

```
REPOSITION_TAPE ( ocount, mcount, mod_oper, mu_desc, reply );
```

Parameter	Type	Description
ocount	LONG_INTEGER	Object offset if the Object-Count modifier is specified; otherwise a record offset—the number of objects or records to skip
mcount	LONG_INTEGER	Tape-mark offset—the number of tape marks to skip; not applicable if the Object-Count modifier is specified
mod_oper	UNSIGNED	Function-modifier values: Rewind value (2) for repositioning from BOT, Object-Count value (4) for use of object offsets, or Reverse value (6) for reverse repositioning
VAR mu_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

When both record and tape-mark offsets are specified, the tape-mark offset is observed first. For example, with the Rewind modifier set, a record offset of 10 (decimal) and a tape-mark offset of 2 would reposition the tape to the eleventh record of the third file.

Indication of success or failure is returned in the MU driver reply packet.

5.3.4 WRITE_TAPE_MARK

The WRITE_TAPE_MARK procedure establishes the end of a logical file by writing a tape mark at the current position.

The packet-level equivalent of WRITE_TAPE_MARK is the IF$MRK function.

The syntax for calling the procedure is as follows:

```
WRITE_TAPE_MARK ( mu_desc, reply );
```

Parameter	Type	Description
VAR mu_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

Indication of success or failure is returned in the MU driver reply packet.

5.3.5 REWIND_TAPE

The REWIND_TAPE procedure requests that the tape be rewound to the BOT. It is logically equivalent to doing a REPOSITION_TAPE with an offset of 0 and the Rewind modifier set.

The packet-level equivalent of REWIND_TAPE is the IF$RWD function.

The syntax for calling the procedure is as follows:

```
REWIND_TAPE ( mu_desc, reply );
```

Parameter	Type	Description
VAR mu_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

Indication of success or failure is returned in the MU driver reply packet.

5.4 Pascal I/O Procedure Interface

To perform standard Pascal I/O to a TMSCP tape controller, you must open a file. Opening the file associates a Pascal file variable with a tape controller unit. Invoke the OPEN procedure as follows:

OPEN (filvar, 'MUc0:', ...)

where:

- filvar is a Pascal file variable.
- c is a controller identifier (A, B, ...).

For example, 'MUA0:' would specify unit 0 of the first TMSCP controller (A) listed in the MU prefix file.

The OPEN statement causes the Pascal OTS to send a packet-level open request to the ACP, which returns a unit number and a driver request semaphore ID to the OTS. Subsequent I/O requests are sent directly to the driver by the OTS with no further ACP involvement.

In carrying out subsequent input, output, CLOSE, or PURGE operations on TMSCP units, the Pascal OTS uses the following packet-level driver functions:

- Read Logical (IF$RDL)
- Write Logical (IF$WTL)
- Close (IF$CLS)
- Purge (IF$PRG)

The appropriate request packets are sent to the driver only when necessary for completion of a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the MU driver. Other Pascal statements unconditionally cause the OTS to issue send requests; examples are BREAK, which generates a Write Logical (IF$WTL), and CLOSE, which generates a Close (IF$CLS) request (normally preceded by a Write Logical, unless BREAK immediately precedes CLOSE).

Pascal functions that report the characteristics of MU-driver-supported hardware are provided in the file GETSET.PAS on the MicroPower/Pascal distribution kit. Those functions issue Get Characteristics (IF$GET) request packets to the driver.

The following packet-level driver functions cannot be performed with normal Pascal I/O statements or GETSET functions:

- Reposition Tape (IF$REP)
- Write Tape Mark (IF$MRK)
- Rewind Tape (IF$REW)

To perform these functions, use the Pascal support routines for nonfile I/O (see the preceding section), use the request/reply packet interface directly, or write Pascal procedures that take a user-specified file variable and send the appropriate request packets to the driver. (The Get Characteristics procedures in GETSET.PAS demonstrate the latter approach.)

5.5 Request/Reply Packet Interface

The packet-level functions provided by the TMSCP driver are listed below by symbolic and decimal function code:

Code	Function
IF$RDL (1)	Read Logical
IF$WTL (4)	Write Logical
IF$GET (7)	Get Characteristics
IF$ONY (8)	Bypass Only—for internal use only
IF$BYP (9)	Bypass—for internal use only
IF$INT (10)	Initialize Port—for internal use only
IF$REP (11)	Reposition Tape
IF$MRK (12)	Write Tape Mark
IF$RWD (13)	Rewind Tape

If a request is received for an Open (IF$LOK or IF$ENT), a Close (IF$CLS), or a Purge (IF$PRG), the driver returns an illegal function status code (ES$IFN), which the ACP (Open) or OTS (Close/Purge) interprets to mean that no device-dependent processing was required for that operation.

Note

The MACRO–11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the MU driver are shown below by symbolic code and bit position:

Code	Function
FM$BSM (bit 13)	Signal binary/counting semaphore
FM$INH (bit 15)	Inhibit retries on error

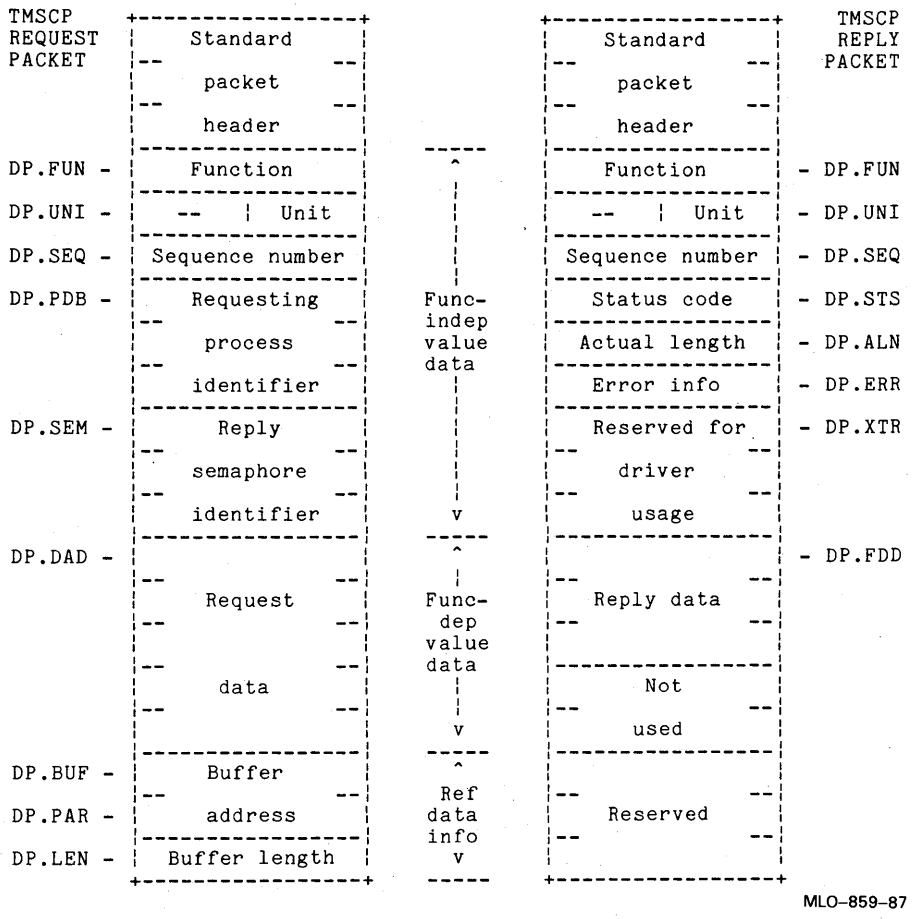
The MU driver consists of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each configured controller. I/O requests for a particular controller are sent (using a Pascal SEND or a MACRO–11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The request queue name and number of supported units for MU driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
TMSCP	$MUc	1	0

The letter c in a queue name represents a controller designation (A, B, ..., as specified in an MU driver prefix file).

The general format of the TMSCP request and reply packets is shown below:



The function-independent portions of the packets shown above are described in Section 1.3, Request/Reply Packet Interface. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of TMSCP driver function.

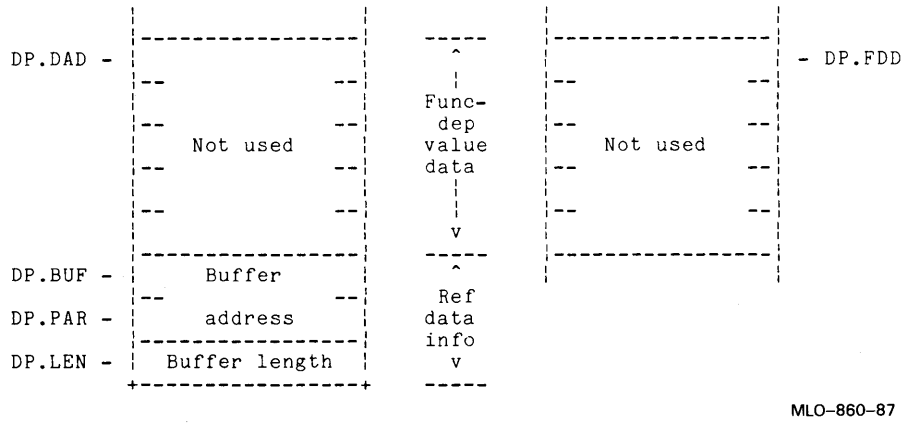
Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

5.5.1 Read and Write Functions

A TMSCP read or write operation transfers data between the user's buffer and the current tape position.

The following are function-dependent portions of the MU read or write request and reply packets:



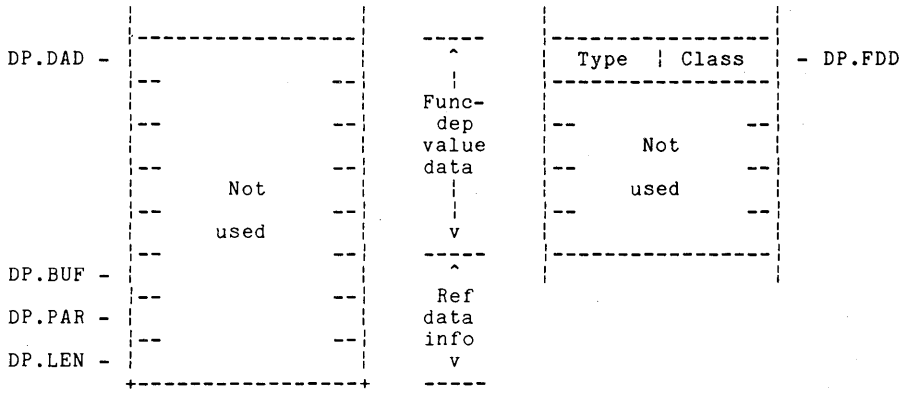
MLO–860–87

The buffer-length value determines the length, in bytes, of the data transfer.

5.5.2 Get Characteristics Function

The MU Get Characteristics function returns the codes for TMSCP device class and type in the function-dependent portion of the reply message.

The following are function-dependent portions of the MU Get Characteristics request and reply packets:



MLO-861-87

In the reply information above:

- Class is DC$TAP for tape device class.
- Type is MT$MU for TMSCP tape device type.

5.5.3 Reposition Tape Function

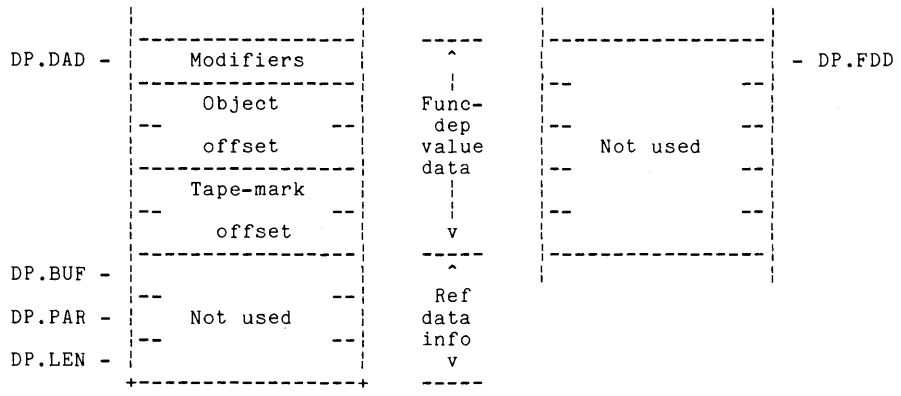
The Reposition Tape function requests that the tape be repositioned to a point that is specified either by use of a generic object count or by use of record and tape-mark counts.

Note

For tape-specific operations, the relevant units are records, tape marks (which indicate the end of a logical file), and objects (a context-dependent term for either records or tape marks).

Depending on user-specified modifiers, the tape can be repositioned to an offset forward or backward from the current position or forward from the BOT.

The following are function-dependent portions of the MU Reposition request and reply packets:



MLO-862-87

The modifier word can specify the following:

- Rewind value (2) for repositioning from the BOT
- Object-Count value (4) for use of object offsets
- Reverse value (6) for reverse repositioning

If the Object-Count modifier is set, the object-offset field gives the number of objects (records or tape marks) to skip, and the tape-mark-offset field is ignored. Otherwise, the object-offset and tape-mark-offset fields give the numbers of records and tape marks to skip. In the latter case, the tape-mark offset is observed before the record offset. For example, with the Rewind modifier set, a record offset of 10 (decimal) and a tape-mark offset of 2 would reposition the tape to the eleventh record of the third file.

5.5.4 Write Tape Mark Function

The Write Tape Mark function establishes the end of a logical file by writing a tape mark at the current position.

The function-dependent portions of the Write Tape Mark request and reply packets are not used.

5.5.5 Rewind Tape Function

The Rewind Tape function requests that the tape be rewound to the BOT. It is logically equivalent to doing a Reposition with an offset of 0 and the Rewind modifier set.

The function-dependent portions of the MU Rewind request and reply packets are not used.

5.6 Status Codes

If an error is detected during an I/O operation by a tape device or the MU driver, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error is detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply message.

The MU driver returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O request canceled or port reinitialized
ES$BOT	HARD_IO	Beginning of tape encountered
ES$CTL	HARD_IO	Controller error, formatter error, or position lost
ES$DRV	HARD_IO	Drive error
ES$IBN	HARD_IO	Invalid block number
ES$IVD	HARD_IO	Data error
ES$IVP	HARD_IO	Invalid command, host buffer access error
ES$NXU	HARD_IO	Nonexistent unit
ES$OFL	HARD_IO	Device off line
ES$OVF	HARD_IO	Data overflow, record data truncated
ES$UNS	HARD_IO	Unsafe volume
ES$WLK	HARD_IO	Write-protected unit
ES$EOF	SOFT_IO	Tape mark encountered
ES$IFN	SOFT_IO	Illegal function

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO–11 users.

Note

Not listed above are exception codes for OTS-detected I/O errors or for kernel-detected errors that the driver raises rather than passing back to the requesting process. OTS-detected I/O errors are listed in Chapter 9 of the *MicroPower/Pascal Language Guide.*

5.7 MU Driver Prefix File

Figure 5–1 shows the TMSCP tape driver prefix module. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The DRVCF$ macro contains a field for the number of controllers on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced by the driver. It gives the controller name, number of units (1), CSR and vector addresses, and unit number (0). The interrupt vectors must also be specified in the system configuration file, using the DEVICES macro.

The MU$IPR, MU$PPR, and MU$HPR definitions specify the initialization and request-handling software priorities for the driver process and the hardware interrupt priority for the controller(s).

Figure 5–1: TMSCP Tape Driver Prefix File (MUPFX.MAC)

```
.title   MUPFX   - TMSCP Micro PDP-11 driver prefix module

;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1983, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS
;   RESERVED.

   .mcall   drvcf$
   .mcall   ctrcf$

MU$PPR   ==   175.      ; Process priority
MU$HPR   ==   4         ; MSCP hardware priority
MU$IPR   ==   250.      ; Process initialization priority

   drvcf$   dname=MU,nctrl=1
   ctrcf$   cname=A,nunits=1.,csrvec=<174500,300>,units=<0:0>

   .end
```

Chapter 6
Parallel Line Drivers

This chapter describes the use of the MicroPower/Pascal parallel line drivers, which support I/O operations on devices connected through parallel line interfaces. The parallel line drivers support the interfaces listed below:

Driver	Supported Interfaces
XA	DRV11–J 64-bit parallel interface (four 16-bit ports)
YA	DRV11 16-bit parallel interface
YB	DRV11–B DMA interface
YF	SBC–11/21 8255 PIO interface
YK	KXT11–CA/KXJ11–CA 8-bit parallel ports (16-bit if linked) 4-bit special-purpose I/O ports 16-bit counter/timers

The supported devices interface parallel lines to a MicroPower/Pascal target processor so that block data transfers can be performed.

DRV11–B block transfers use direct memory access, minimizing processor involvement. It is also possible to coordinate the YK driver with the KXT11–CA and KXJ11–CA DMA (QD) driver to effect DMA block transfers through a KXT11–CA or KXJ11–CA parallel port.

Note

Unlike other MicroPower/Pascal drivers, the DRV11 (YA) driver is not included in the driver object libraries. It is distributed as two source modules—the driver proper (YADRV.PAS) and the driver prefix file (YAPFX.PAS). It is available for applications that require it and/or as a base for editing and/or as an example of a device driver coded in Pascal.

The DRV11–J (XA) driver is distributed in both object and source form; source modules XADRV.MAC and XAPFX.MAC are available as a base for editing.

6.1 Parallel Line Driver Features and Capabilities

The parallel line drivers support block-mode read and write operations and returning of device characteristics.

Read and write operations transfer a specified number of data bytes to or from the caller's buffer. For the DRV11–J, DRV11, and DRV11–B, data is transferred by word. For the SBC–11/21 PIO interface, data is transferred by byte. For the KXT11–CA and KXJ11–CA PIO interface, data is transferred by byte or (if two ports are linked) by word. An even number of bytes must be specified for devices that do word transfers.

KXT11–CA and KXJ11–CA parallel port read/write operations have pattern recognition capabilities, by which a transfer can be made to terminate when a specified pattern is found or a search limit is reached.

Get Characteristics operations report standard device characteristics, including device class and type.

In addition to read, write, and Get Characteristics operations, most of the drivers support operations that are specific to the interfaces they support.

The XA driver supports enabling or disabling of special interrupt functions on DRV11–J port A. In the standard, factory-jumpered DRV11–J configuration, port A provides 12 sense lines (0 to 11), each capable of generating unique interrupt requests. These lines are particularly useful for monitoring specific points in your application environment (for example, in process control applications). The remaining four bits in port A monitor the four user-reply signals (USER RPLY A to USER RPLY D) on the DRV11–J's two I/O connectors. These four bits are capable of generating interrupt requests in response to events in or requests by your hardware. In addition to the interrupt functions of port A, all 16 lines can be used for I/O operations as provided through ports B, C, and D.

The selection of low-active or high-active signals for generating interrupt requests and the selection of rotating or fixed priority for interrupts within each of two 8-bit interrupt groups are made in the XA driver prefix file. See Section 6.8.1 for details.

The DRV11–B Set Characteristics function establishes internal default CSR settings to be used for subsequent DMA transfers. Initial default settings of those bits are determined in the YB driver prefix file. The settable bits include bits that control the initiation of DMA transfers and three function bits available for user-defined purposes.

The KXT11–CA/KXJ11–CA PIO (YK) driver supports pattern recognition on PIO reads and writes, DMA transfers, use of a KXT11–CA or KXJ11–CA parallel port (in coordination with the KXT11–CA/KXJ11–CA DMA driver), and the setting, reading, and clearing of the KXT11–CA and KXJ11–CA counter/timers. Many types of I/O operations are possible, depending on how the parallel ports and counter/timers are configured and programmed. See Section 6.8.5 for information on different configurations. Bidirectional mode is not supported.

6.2 Performing Parallel I/O

For most MicroPower/Pascal applications—except KXT11-CA and KXJ11-CA target applications—you perform parallel I/O by invoking Pascal I/O procedures that open files for parallel line data and then input or output the data, in accordance with the rules for Pascal I/O. The Pascal I/O procedures—OPEN, GET, WRITE, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide*.

File-oriented operations on the KXT11-CA or KXJ11-CA parallel ports are allowed but are of limited usefulness, because the YK pattern setting, DMA transfer, and counter/timer functions cannot be performed by Pascal I/O procedures. For most MicroPower/Pascal applications, you perform KXT11-CA or KXJ11-CA PIO by invoking Pascal support routines—YK_PORT_READ, YK_SET_PATTERN, YK_READ_TIMER, and so forth. Those routines provide high-level nonfile access to the KXT11-CA or KXJ11-CA parallel ports and counter/timers. (Optionally, you can modify the pattern setting and counter/timer routines to accept a file variable.) The YK support routines issue Pascal send requests to the request queue semaphore of the YK driver. The routines are described in Section 6.4.2.

Note

The DRV11-J (XA) driver sense line Enable and Disable operations also cannot be performed with Pascal I/O procedures. See Section 6.3 for more information on such operations.

In addition to invoking the Pascal I/O procedures, or KXT11-CA or KXJ11-CA support routines, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the parallel-line controller interrupt vector addresses
2. Edit the parallel line driver prefix file to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, number of controller units and their identifying numbers (0, 1, ...)
   - Hardware interrupt priority
   - Other hardware/interface characteristics, such as DRV11-J sense-line signal and priority settings, DRV11-B default CSR settings, or KXT11-CA or KXJ11-CA parallel port and timer port attributes
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - Parallel line driver process
   - [For non-file-oriented KXT11-CA or KXJ11-CA PIO:] Pascal KXT11-CA and KXJ11-CA PIO support routines (from kit files YK.PAS and YKINC.PAS)
   - [For parallel line file OPEN:] Ancillary control process (ACP)

- Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any file-oriented support routines you opt to include (see kit files GETSET.PAS and GSINC.PAS)

Note

In addition to the KXT11-CA and KXJ11-CA PIO support routines and file-oriented support routines in the kit files mentioned above, the MicroPower/Pascal distribution kit provides support routines for non-file-oriented, non-interrupt parallel I/O on the SBC-11/21 board. Those routines, discussed below and in Section 6.4, do not require the OTS file-service routines, the ACP, or the YF driver.

For more information on setting up your application software for parallel I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 6.8 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

When a module that contains Pascal I/O procedure invocations is built into your application, Pascal OTS routines for file service are linked to the module. The OTS file routines perform all Pascal operations on files, including file opening, input, and output. In particular, they perform the necessary low-level processing of high-level operations such as OPEN and WRITE. Thus, the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to driver or ACP queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores—are concealed from the Pascal user.

Alternatives to using the Pascal I/O procedures for parallel I/O exist, but require more effort. (The PIO support routines for KXT11-CA, KXJ11-CA, and SBC-11/21 applications were mentioned previously in this section.) You can:

- Issue your own Pascal or MACRO-11 packet-level requests to the ACP and the driver, bypassing the OTS file routines (lower-level file system access).
- [For KXT11-CA, KXJ11-CA or SBC-11/21 PIO:] Invoke Pascal routines that support non-file-oriented parallel I/O (high-level nonfile access).
- Issue your own Pascal or MACRO-11 packet-level requests to the driver, bypassing the OTS file routines, the ACP, and the Pascal support routines for nonfile I/O (low-level nonfile access).

The following sections describe the Pascal I/O procedure interface to the parallel line drivers, the Pascal support routines, the lower-level request/reply packet interface, the status codes that can be returned to users of any interface, and the parallel line driver prefix files.

6.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O to a parallel line, you must open a file. Opening the file associates a Pascal file variable with a parallel controller unit. Invoke the OPEN procedure as follows:

```
OPEN (filvar, 'ddcu:', ...)
```

where:

- filvar is a Pascal file variable.
- dd is the driver identifier (XA for DRV11-J, YA for DRV11, YB for DRV11-B, YF for SBC-11/21 PIO, YK for KXT11-CA and KXJ11-CA parallel ports—KXT11-CA and KXJ11-CA counter/timers not accessible).
- c is a controller identifier (A, B, ...).
- u is a controller unit number (0, 1, ...).

For example, 'XAA1:' would specify the second unit (1) of the first DRV11-J controller (A) listed in the XA driver prefix file.

Note

The DRV11 (YA), SBC-11/21 PIO (YF), and KXT11-CA and KXJ11-CA PIO (YK) drivers do not support multiple controllers; specify A for the controller identifier.

The number of units supported for each parallel line controller is shown below:

Controller	Number of Units	Numbering
DRV11-J	[For read/ write:] 1-4 [For packet-level Enable/ Disable:] 1-12	0 through 3 for ports A through D 4 through 15 for port A lines 0 through 11
DRV11	1	0
DRV11-B	1	0
SBC-11/21 PIO	1-2	0 and 1 for ports A and B
KXT11-CA or KXJ11-CA PIO	1-6	0 through 2 for ports A through C and 3 through 5 for timers 1 through 3 (timer units cannot be accessed by Pascal I/O procedures)

The number of units actually configured for each controller and their unit numbers must be specified in a parallel line driver prefix file.

The OPEN causes the Pascal OTS to send a packet-level open request to the ACP, which returns a unit number and a driver request semaphore ID to the OTS. Subsequent I/O requests are sent directly to the driver by the OTS, with no further ACP involvement.

In carrying out subsequent input, output, CLOSE, or PURGE operations on parallel line controller units, the Pascal OTS uses the following packet-level driver functions:

- Read Logical (IF$RDL)
- Write Logical (IF$WTL)
- Close (IF$CLS)
- Purge (IF$PRG)

The appropriate request packets are sent to the driver only when necessary for completion of a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the parallel line driver. Other Pascal statements unconditionally cause the OTS to issue send requests; examples are BREAK, which generates a Write Logical (IF$WTL), and CLOSE, which generates a Close (IF$CLS) request (normally preceded by a Write Logical, unless BREAK immediately precedes CLOSE).

Pascal Get and Set Characteristics functions that report or alter the characteristics or status of supported parallel interfaces are provided in the file GETSET.PAS on the MicroPower/Pascal distribution kit. Those functions issue Get and Set Characteristics (IF$GET and IF$SET) request packets to the driver.

The following packet-level driver functions cannot be performed with normal Pascal I/O statements or GETSET functions:

- XA Enable (IF$ENA)
- XA Disable (IF$DSA)
- YK Set Pattern (IF$YKP)
- YK DMA Read (IF$YKR)
- YK DMA Write (IF$YKW)
- YK DMA Complete (IF$YKE)
- YK Set Timer (IF$YKS)
- YK Clear Timer (IF$YKU)
- YK Read Timer (IF$YKT)

To perform those functions, use Pascal support routines (available for KXT11-CA or KXJ11-CA PIO only), use the request/reply packet interface directly, or write Pascal procedures that take a user-specified file variable (or queue semaphore ID) and send the appropriate request packets to the driver. (The Get/Set Characteristics procedures in GETSET.PAS demonstrate the last approach.)

6.4 Pascal Support Routines

The following support routines, written in Pascal and independent of the file system, provide alternative high-level interfaces to the SBC–11/21 and KXT11–CA or KXJ11–CA PIO hardware:

- SET_PIO_MODE procedure
- READ_PIO function
- WRITE_PIO procedure
- YK_PORT_READ function
- YK_PORT_WRITE function
- YK_SET_PATTERN function
- YK_SET_TIMER function
- YK_READ_TIMER function
- YK_CLEAR_TIMER function

The first three routines support SBC–11/21 8255 PIO in noninterrupt mode and are independent of the SBC–11/21 PIO driver. They are used to set up and access the PIO ports directly from a user process, using programmed I/O.

The remaining routines support KXT11–CA or KXJ11–CA PIO via the YK driver. The KXT11–CA and KXJ11–CA routines use all the YK packet-level functions, except the following:

- Get Characteristics (IF$GET)
- DMA Read (IF$YKR)
- DMA Write (IF$YKW)
- DMA Complete (IF$YKE)

Note

A non-file-oriented Get Characteristics function is provided in the distribution kit file GETSET.PAS.

See Section 6.4.2.4 for details on the use of a KXT11–CA/KXJ11–CA parallel port for DMA transfers.

The following sections describe the Pascal functions for non-file-oriented parallel I/O on the SBC–11/21 and the KXT11–CA or KXJ11–CA.

6.4.1 SBC–11/21 PIO Support Routines

The SBC–11/21 PIO routines allow you to set up and access the SBC–11/21 PIO ports directly from a user process, with no driver involvement. (This differentiates those routines from most MicroPower/Pascal support routines, which send packet-level requests to drivers.) The routines support the SBC–11/21 on-board 8255 PIO in mode 0 (noninterrupt mode). In mode 0 there are two 8-bit data ports (A and B) and a third dual 4-bit port (C). The lower half of port C is permanently connected as an input. Ports A and B and the upper half of port C can be used as either input or output, as determined by wire-wrap jumpers on the SBC–11/21 board.

The following files on the MicroPower/Pascal distribution kit are required for using the functions:

Name	Description
YFDRVP.PAS	SBC–11/21 noninterrupt PIO routine source module
YFDRVI.PAS	SBC–11/21 noninterrupt PIO routine include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include file must be included in the program at compile time.

The following SBC–11/21 data structures, defined in YFDRVI.PAS, are referenced throughout the rest of this section:

```
TYPE
    port_sel = (port_a, port_b, port_c_low, port_c_high);

    mode_sel = (Ainput_Binput_Cinput, Ainput_Binput_Coutput,
                Ainput_Boutput_Cinput, Ainput_Boutput_Coutput,
                Aoutput_Binput_Cinput, Aoutput_Binput_Coutput,
                Aoutput_Boutput_Cinput, Aoutput_Boutput_Coutput);
```

6.4.1.1 SET_PIO_MODE

The SBC–11/21 procedure SET_PIO_MODE sets one of eight modes, each of which represents a different combination of input/output settings for the three ports.

The syntax for calling the procedure is as follows:

```
SET_PIO_MODE ( mode );
```

Parameter	Type	Description
mode	mode_sel	Mode selected—one of the eight possible combinations of input/output settings for ports A, B, and C (high)

6.4.1.2 WRITE_PIO

The SBC–11/21 procedure WRITE_PIO writes a value to a user-specified port.

The syntax for calling the procedure is as follows:

```
WRITE_PIO ( port, outdat );
```

Parameter	Type	Description
port	port_sel	Port selected
outdat	INTEGER	Value to be written—must be in 8-bit or 4-bit value range as appropriate for the port

6.4.1.3 READ_PIO

The SBC-11/21 function READ_PIO reads a user-specified port and returns a value of type INTEGER.

The syntax for calling the function is as follows:

```
READ_PIO ( port )
```

Parameter	Type	Description
port	port_sel	Port selected

6.4.2 KXT11-CA/KXJ11-CA PIO and Counter/Timer Support Routines

Each KXT11-CA or KXJ11-CA PIO routine allocates an I/O packet, fills it with information based on the function parameters, and sends it to the YK driver.

If a reply semaphore is provided in the call, the function returns immediately after sending the driver request. When the operation is complete, the driver sends a standard device driver reply via the specified semaphore. (The driver reply is described in Section 6.5.) The completion status returned in the reply packet must be processed by a routine that is waiting on the semaphore. For PIO read/write operations, the routine that waits on the semaphore must also process the actual-length information in the packet.

If no reply semaphore is provided, the function waits for the driver reply before returning to the caller.

The KXT11-CA and KXJ11-CA PIO functions allow you to issue multiple requests for a single KXT11-CA or KXJ11-CA parallel port. Thus, you can set up a double-buffering type of operation, with a second buffer starting to be filled/sent while a first buffer is returned/acknowledged to the caller.

In addition, pattern-matching commands can be issued in conjunction with the PIO transfer commands. For example, consider a case in which a buffer is to be filled until a special character is received and then a second buffer is to be filled until a different special character is received. The function calls to accomplish are a YK_SET_PATTERN, a YK_PORT_READ, a second YK_SET_PATTERN, and a second YK_PORT_READ. All four calls can be issued without waiting for a reply from any of them. You can continue processing until signaled that the first portion has been received; then the device driver can continue receiving the second portion while you are processing the first.

The following files on the MicroPower/Pascal distribution kit are required for using the functions:

Name	Description
YK.PAS	KXT11-CA and KXJ11-CA PIO and counter/timer function source module
YKINC.PAS	KXT11-CA and KXJ11-CA PIO, C/T function and driver packet include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

The following data type from YKINC.PAS is referenced throughout this section; it defines the YK unit numbers for the support routine interface:

```
TYPE
   UNIT_NUMBER = (
     PORT_A ,      { Port A }
     PORT_B ,      { Port B }
     PORT_C ,      { Port C }
     TIMER_1 ,     { Timer 1 }
     TIMER_2 ,     { Timer 2 }
     TIMER_3 ) ;   { Timer 3 }
```

6.4.2.1 YK_PORT_READ

The YK_PORT_READ function transfers data from a parallel port to a KXT11-CA or KXJ11-CA buffer and returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

The syntax for calling the function is as follows:

```
YK_PORT_READ ( port_num, buffer, byte_count, reply, match_rst, seq_num )
```

Parameter	Type	Description.
port_num	UNIT_NUMBER	Number of port to be read from.
VAR buffer	UNIVERSAL	Data buffer address; if omitted, a "signal semaphore only" operation is implied, and the byte count must be 0.
VAR byte_count	UNSIGNED	Number of bytes to be read. If in pattern-match mode, the count specifies an upper limit instead of an actual count. If the limit is reached before the pattern is matched, an error is reported. When the pattern is found, the read terminates, and BYTE_COUNT is set to the actual number of bytes read. In pattern-match mode, the last byte in the buffer will be the one that matched. BYTE_COUNT is not returned if the reply parameter is provided.

Parameter	Type	Description.
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL.
match_rst	BOOLEAN	Optional parameter that, if TRUE, causes a previously set pattern mode to be reset at the end of the read command; default is FALSE.
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet).

If no reply parameter is provided, the function sets the parameter BYTE_COUNT to the count of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the YK driver reply packet.

6.4.2.2 YK_PORT_WRITE

The KXT11-CA/KXJ11-CA function YK_PORT_WRITE transfers data from a KXT or KXJ buffer to a parallel port and returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

The syntax for calling the function is as follows:

```
YK_PORT_WRITE ( port_num, buffer, byte_count, reply, match_rst, seq_num )
```

Parameter	Type	Description
port_num	UNIT_NUMBER	Port number to be written to.
VAR buffer	UNIVERSAL	Data buffer address.
VAR byte_count	UNSIGNED	Number of bytes to be written. If pattern-match mode is enabled on at least one of the output lines, the byte count specifies an upper limit instead of an actual length. If the limit is reached before the pattern is matched, an error is reported. When the pattern is found, the write terminates, and BYTE_COUNT is set to the actual number of bytes that were written. BYTE_COUNT is not returned if the reply parameter was provided.
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL.
match_rst	BOOLEAN	Optional parameter that, if TRUE, causes a previously set pattern mode to be reset at the end of the write command; default is FALSE.
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet).

If no reply parameter is provided, the function sets the parameter BYTE_COUNT to the count of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the YK driver reply packet.

6.4.2.3 YK_SET_PATTERN

The KXT11-CA/KXJ11-CA function YK_SET_PATTERN controls the pattern-recognition features of the peripheral processor. Specifically, it sets pattern-match mode on parallel port A or B. The setting of pattern-match mode affects subsequent operation of the YK_PORT_READ and YK_PORT_WRITE functions. In pattern-match mode, a read or a write operation terminates only when a specified pattern is found in the data or when the user-imposed search limit in the read or write request (BYTE_COUNT) is reached.

The YK_SET_PATTERN function returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

To use pattern matching, you must specify "PAT=YES" for port A or B in the prefix file port-configuration (YKCP$) macro.

The syntax for calling the function is as follows:

```
YK_SET_PATTERN ( port_num, mode, reply, patp, patt, patm, pt_buf, seq_num )
```

Parameter	Type	Description
port_num	UNIT_NUMBER	Port number.
mode	PAT_MOD_ENTRY	Pattern modifier bits; AND_MODE, the default, indicates that all specified pattern bits must match. OR_MODE indicates that only one of the specified pattern bits must match. WAIT_MATCH sets wait-for-pattern-match mode. PAT_RESET causes the pattern mode to be reset after a command. Note that OR_MODE and AND_MODE are the only modifiers that are mutually exclusive; all other combinations are valid.
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL.

Parameter	Type	Description
patp patt patm	BYTE_RANGE BYTE_RANGE BYTE_RANGE	The PATP, PATT, and PATM parameters collectively define the match pattern for the specified port. Each bit (0–7) in a PATP, PATT, or PATM specification corresponds to a bit (0–7) in the match pattern; that is, bit n of the match pattern is defined by the nth bits of PATP, PATT, and PATM. For each match pattern bit, PATP supplies pattern polarity information; PATT, pattern transition information; and PATM, pattern mask information. For details on the significance of PATP/PATT/PATM bit combinations, see the table and the example below. The default value for each parameter is 0.
pt_buf	YKBUF_PT	Optional buffer pointer used only in wait_match mode; default is NIL. If omitted for wait_match mode operation, one byte of data—two if ports are linked—will be returned in first word of function-dependent portion of YK driver reply packet.
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet).

The PAT_MOD_ENTRY, PATTERN_MODS, and YKBUF_PT data types, from YKINC.PAS, are shown below:

```
TYPE
   Pattern_mods = (     { Pattern Function Modifiers }
     nu_3 ,             { - not used }
     nu_4 ,             { - not used }
     pat_reset ,        { - reset pattern at end }
     and_mode ,         { - AND pattern mode }
     or_mode ,          { - OR pattern mode }
     wait_match ) ;     { - Wait until match mode }

   Pat_mod_entry = PACKED SET OF Pattern_mods ;

   YKBUF_PT = ^ UNSIGNED ;  { data buffer pointer }
```

The pattern specification for each bit of the match pattern is defined as follows:

PATP	PATT	PATM	Event Recognized
x	0	0	Bit masked off—no event recognized
x	1	0	Any transition
0	0	1	Logical 0 state
1	0	1	Logical 1 state
0	1	1	Logical 1 to logical 0 transition
1	1	1	Logical 0 to logical 1 transition

Note

Do not specify more than one bit to detect transitions if you specify AND_MODE.

For example, to set a pattern of bits 0 to 3 = 1 AND bits 5 and 6 = 0 AND bit 7 = logical 1 to logical 0 transition AND bit 4 ignored, you would pass the following bits in the PATP, PATT, and PATM parameters:

Bit	PATP	PATT	PATM
0	1	0	1
1	1	0	1
2	1	0	1
3	1	0	1
4	0	0	0
5	0	0	1
6	0	0	1
7	0	1	1

The following function call would set the desired pattern:

```
YK_SET_PATTERN (port_num := PORT_A,
                mode := [ and_mode ],
                patp := %O'17',
                patt := %O'200',
                patm := %O'357')
```

6.4.2.4 KXT11-CA/KXJ11-CA PIO DMA Process

If you want to perform DMA transfers via a KXT11-CA/KXJ11-CA parallel port, you must first set up and send a DMA Read or a DMA Write request packet to the YK driver and wait for the reply. If the reply indicates normal status, you then send a DMA transfer command to the DMA (QD) driver; otherwise, you report a software exception. You must wait for each request to complete, since only one PIO DMA operation can be in progress at a time. After the DMA transfer completes, you send a DMA Complete request to the YK driver, which unlocks the queue of requests for that port.

Observe the following guidelines when performing DMA I/O on a KXT11-CA or KXJ11-CA parallel port:

- Use KXT11-CA or KXJ11-CA DMA channel B (QD unit 1) for the QD requests. Channel B is linked to the timer/counter (KXT11-CA or KXJ11-CA PIO) chip when you install the jumper to configure the DMA request lines. (See Section 6.3.12 of the *SBC-11/21 Single-Board Computer User's Guide* for details on installing the jumper or see the *KXJ11-CA Single-Board Computer User's Guide.*)
- Specify "DMA=YES" for the KXT11-CA or KXJ11-CA PIO port in the YK prefix file port-configuration (YKCP$) macro.
- Use KXT11-CA/KXJ11-CA PIO port A (YK unit 0). Line C1 is connected to the DMA request line and therefore is not available for handshake for port B. (This means that port B, if used, must be configured as a bit port.) In the prefix file, line C1 must be set up as an inverted output so that it works correctly with the DMA request line.
- In the QD transfer request, specify wait-for-request mode and byte mode. Also, specify the address of the data CSR for YK port A as 177033 (octal), not 177032 (octal). This is necessary because the DMA chip addresses bytes in memory in a way different from typical LSI-11 hardware; the chip's high-order byte is LSI-11's low-order byte, and vice-versa. Specify that the data CSR for YK port A is in the I/O page (DMA$ IO option). Use an even address for the other address.
- After a transfer, the data in the destination buffer has each byte reversed, again because of the way the DMA chip addresses bytes. For example, "abcd" in a source buffer becomes "badc" in the destination buffer. Therefore, you should reverse the bytes in each word before transmitting the data or after receiving the data. (A procedure for reversing the bytes is shown in the following example.)

Figure 6-1 shows a sample program that uses the YK and QD drivers to perform DMA I/O on a KXT11-CA/KXJ11-CA parallel port. Data is transferred from the parallel port to local memory on the KXT11-CA/KXJ11-CA. Note that the program calls the local procedure REVERSE_BUF after data is received to reverse the bytes in the data buffer.

Figure 6-2 shows the appropriately modified YK driver prefix file. (The YK prefix file is described in Section 6.8.5.)

Figure 6–1: KXT11–CA/KXJ11–CA PIO DMA Sample Program

```
{Notes:  1. Can only use Parallel Port A for byte I/O when using the
            DMA chip, Port B must be a bit port.
         2. Can only use DMA unit 1 for I/O to YK port.}

[ SYSTEM(MICROPOWER), PRIORITY(50),
  DATA_SPACE(2100), STACK_SIZE (200)] PROGRAM yktio;

{$NOLIST}
%include 'mutex.pas'              { mutex procedures }
%include 'ykinc.pas'     { get the YK data structures and
                                    handler interface }
%include 'qdinc.pas'     { DMA include files }
%include 'exc.pas'                { Get exception processing, it picks
                                    up escode.pas }
{$LIST}

CONST
  BUFSIZE = 30;                          {size of buffer}
  INITBUF = 'ABCDEFGHIJKLMN OPQRSTUVWXYZ123';  {data for input buffer}
VAR
  protect_mutex : mutex;          {screen access protection}
  inbuf : PACKED ARRAY[1..BUFSIZE] OF CHAR;  {buffers}
  inbuf_size : UNSIGNED;                {size of buffers}
  in_reply_descriptor : STRUCTURE_DESC; {queue semaphore
                                         descriptors for replies}
  in_reply_packet : YK_REPLY;  {reply packet from driver}
  in_status : UNSIGNED; {Dummy status for return by Yk_port_read
                                 and yk_port_write functions}
  address_1,address_2 : DMA$ADDRESS;
  byte_count : DMA$BYTE_COUNT;
  yk_req : yk_port_rqst;
  k : integer;

[INITIALIZE] PROCEDURE YK_INIT ;        {Initialize procedure. Runs once at
                                         startup.}
BEGIN
IF NOT CREATE_QUEUE_SEMAPHORE (DESC := in_reply_descriptor) THEN
       WRITELN ('Create semaphore failed');
END ;

{Main program}
BEGIN

      FOR k := 1 to BUFSIZE DO  {clear the input buffer}
          inbuf[k] := ' ';
      writeln ('Input buffer initialized');

      inbuf_size := bufsize;
```

```
{set up dma request to DMA port}
      with yk_req do
         BEGIN
            oper := dma_read;
            funct_mods := [];
            unit_num := port_A;
            reply_sem := in_reply_descriptor.id;
          END;
      send ( name := '$YKA  ',                    {set up in DMA read mode}
             val_data := yk_req,
             val_length := size (yk_port_rqst) );
      receive (val_data := in_reply_packet,       {wait for set up to
                                                   complete}
               val_length := size (in_reply_packet),
               desc := in_reply_descriptor);

         IF in_reply_packet.status <> es$nor THEN
         writeln ( 'Error setting up DMA read');

       address_1 := DMA$NORM_IBUS_ADDRESS;
{      address_1.high := %o'77';
       Optional since already saying on the i/o page }
       address_1.low := %o'177033';
       address_1.inc := DMA$NOINC;
       address_1.bm := DMA$BYTE;
       address_1.io := DMA$IO;     {Address is on the local i/o page}
       address_1.wfr := dma$wfr;
       address_1.adrtyp := dma$physical; {Address is a physical address}

       address_2 := DMA$NORM_IBUS_ADDRESS;
       address_2.low := (ADDRESS(inBUF))::UNSIGNED;

 {It's a read operation since destination is a plain buffer}
       BYTE_COUNT := $DMA_TRANSFER (       { transfer... }
         UNIT := 1,                        { on this unit }
         SOURCE := address_1,              { from DMA port }
         DEST := address_2,                { to the other local buffer }
         COUNT := inbuf_SIZE );            { this much }
       IF BYTE_COUNT = 0
       THEN
         REPORT(EXC_TYPE:=[RESOURCE],EXC_CODE:=ES$RSC)
       ELSE
         BEGIN
           writeln ('Read successful ',BYTE_COUNT,' bytes');
           WRITELN (INBUF);
         END;

      yk_req.oper := dma_complete;  {finish DMA transaction}
      send ( name := '$YKA  ',
             val_data := yk_req,
             val_length := size (yk_port_rqst) );
      receive (val_data := in_reply_packet,  {wait for operation to
                                              complete}
               val_length := size (in_reply_packet),
               desc := in_reply_descriptor);

         IF in_reply_packet.status <> es$nor THEN
         writeln ( 'Error finishing up DMA read');
end.
```

Figure 6–2: YK Prefix File for PIO DMA Sample Program

```
        .TITLE  YKPFXS - Parallel I/O and counter/timer Driver Prefix Module
        .ident  /V2.0/
;+
;DEFINE PRIORITIES FOR YK HANDLER
;-
YK$HPR  == 5            ; Hardware priority
YK$IPR  == 250.         ; Initialization priority
YK$PPR  == 180.         ; Process priority

;+
;CALL:  INITIALIZE MACRO
;-
        .MCALL  YKCI$
        YKCI$
;+
; PORT A
;Configure as an input port in byte mode with interlocked handshake,
;get DMA support
;-
        YKCP$  CHAN=A,PTYPE=YK$INP,HSH=YK$INL,DMA=YES

;+
; PORT B
;Port A used with DMA chip, so one Port C bit normally used for Port B
;handshake is unavailable, therefore Port B must be a bit port.
;-
        YKCP$  CHAN=B,PTYPE=YK$BIT
;+
; PORT C
;Interlocked handshake signals for Port A (bits 2 and 3),
;bit 1 is for the DMA request line (MUST BE INVERTED)
;-
        YKCP$  CHAN=C,OUT=<YK$B1!YK$B3>,INV=YK$B1
;+
; TIMER 1
;Not used
;-
;+
; TIMER 2
;Not used
;-
;+
; TIMER 3
;Not used
;-
;+
; END CONFIGURATION
;-
        YKCE$
        .END
```

6.4.2.5 YK_SET_TIMER

The KXT11-CA/KXJ11-CA function YK_SET_TIMER can set a timer to an initial value, trigger a timer after setting it, or set a timer to signal a binary or counting semaphore periodically. This function can be used in conjunction with the YK_READ_TIMER function to time or count real-time events. It returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

Timers are initialized to a value (timer_value, below) and count down to 0. You subtract the current value from the initial value to calculate the number of ticks.

The timers count down at a rate of 2MHz, or one tick every .5 microseconds. At that rate, counting down to 0 from the maximum 16-bit timer value (65536) takes approximately 33 milliseconds. For longer intervals, you can have the timer count from 65536 to 0 several times, or link two timers together to make a 32-bit timer. (To specify the maximum initial value, you should enter 0 rather than its equivalent 65536.) To link two timers together, specify "TLNK=YK$1I2" in the YK prefix file macro YKCT$. (See the example in Section 6.4.2.9.)

The syntax for calling the function is as follows:

```
YK_SET_TIMER ( timer_num, timer_value, mode, reply, bin_sem )
```

Parameter	Type	Description
timer_num	UNIT_NUMBER	Timer number.
timer_value	UNSIGNED	Timer constant (TC) value.
mode	TIMER_MOD_ENTRY	Timer mode; INIT_CONSTANT causes the timer constant (TC) value to be set, TRIGGER causes the timer to be triggered after setup, and CONTIN_CYCLE causes the driver to signal the binary or counting semaphore bin_sem, if specified, and restart the timer after each timeout. If CONTIN_CYCLE is not specified, the timer just counts down once (single cycle), and the driver then replies to the user. If timer mode is omitted, the mode set by the prefix file or the last timer command remains in effect.
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL.
bin_sem	STRUCTURE_DESC_PTR	Optional pointer to an initialized binary or counting semaphore to be signaled on each timer timeout; default is NIL.

The TIMER_MOD_ENTRY and TIMER_MODS data types, from YKINC.PAS, are shown below:

```
TYPE
   Timer_mods = (          { Timer Function Modifiers }
     nu_5 ,                        { - not used }
     nu_6 ,                        { - not used }
     init_constant ,               { - initialize timer constant }
     trigger ,                     { - trigger timer when set up }
     nu_7 ,                        { - not used }
     contin_cycle ) ;              { - continuous cycle mode }

   Timer_mod_entry = PACKED SET OF Timer_mods ;
```

Note

If port C is being used to supply handshake signals while timer 3 is being used as a general-purpose timer, the time constant must be set for timer 3 during initialization and not changed during operation. The reason is that port C is disabled during the setting of timer 3's constant, and therefore the handshake signals also get disabled.

6.4.2.6 YK_READ_TIMER

The KXT11-CA/KXJ11-CA function YK_READ_TIMER reads the current count from a timer/counter and returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

The syntax for calling the function is as follows:

```
YK_READ_TIMER ( timer_num, pt_time, reply )
```

Parameter	Type	Description
timer_num	UNIT_NUMBER	Timer number.
pt_time	YKBUF_PT	Optional pointer to time variable; default is NIL; if pointer omitted, timer count value will be returned in first word of function-dependent portion of YK driver reply packet.
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL.

The YKBUF_PT data type, from YKINC.PAS, is shown below:

```
TYPE
   YKBUF_PT = ^ UNSIGNED ;  { data buffer pointer }
```

6.4.2.7 YK_CLEAR_TIMER

The KXT11-CA/KXJ11-CA function YK_CLEAR_TIMER deactivates a timer/counter that is operating in the continuous mode and returns a completion-status value of type UNSIGNED. See Section 6.6 for a list of completion-status values.

The syntax for calling the function is as follows:

```
YK_CLEAR_TIMER ( timer_num, reply )
```

Parameter	Type	Description
timer_num	UNIT_NUMBER	Timer number
reply	STRUCTURE_DESC_PTR	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL

6.4.2.8 Using Timer/Counters to Count External Pulses

Figure 6-3 shows a sample program that creates an external pulse counter that is unaffected by timing delays caused by software. The hardware is set up to stop the counting process at the instant it presents an interrupt request to the processor.

The example uses timer 1 for counting external pulses and timer 3 for timing the counting interval. Timer 3 causes the software to be signaled and stops timer 1 from counting by shutting off its gate input. Thus, when the software reads the number of counts from timer 1, it will be exact. Timer 3 must be set up to have a one-shot input and run in the noncontinuous mode to accomplish this. To expand timer 1 into a 32-bit counter, you can link timer 2 to it. (See Section 6.4.2.9.)

Figure 6–3: KXT11–CA/KXJ11–CA External Pulse Counter Sample Program

```
[SYSTEM(MICROPOWER)] PROGRAM YK8TIM;

{$NOLIST }
%include 'ykinc.pas'
{$LIST }

CONST
  interval_init = 20000;
  count_init = 0;               {0 corresponds to maximum initial value,
                                 = 65536}
VAR
  yk_io_stat    : unsigned;
  reply_desc    : queue_semaphore_desc;
  reply_pkt     : yk_reply;
  count         : unsigned;
BEGIN
    IF CREATE_QUEUE_SEMAPHORE (DESC := REPLY_DESC) THEN
        BEGIN

            {Set an initial value in the timer}
            {Contin_cycle needed since in one shot mode the driver does
             not reply to the reply semaphore set up by the yk_set_timer
             procedure until the timer counts down to 0 and interrupts.
             Since we are using it as a counter of external pulses, the
             counter should never count down to 0. With contin_cycle,
             the driver replies to indicate the status of the set timer
             request and then signals a binary or counting semaphore if the
             counter counts down to 0. No binary or counting semaphore is
             needed since the counter shouldn't count down to 0 before the
             interval times out.}
            yk_io_stat := YK_SET_TIMER
                                ( TIMER_NUM := timer_1,
                                  TIMER_VALUE := count_init,
                                  MODE := [ init_constant, trigger,
                                            contin_cycle]);

            {Now start the time interval for counting pulses}
            yk_io_stat := YK_SET_TIMER
                                ( TIMER_NUM := timer_3,
                                  TIMER_VALUE := interval_init,
                                  MODE := [ init_constant, trigger],
                                  REPLY := ADDRESS (reply_desc));

            {Wait for the time interval to expire}
            RECEIVE ( VAL_DATA := reply_pkt,
                      VAL_LENGTH := SIZE (yk_reply),
                      DESC := reply_desc);

            {Read the current count and convert to # of pulses}
            yk_io_stat := YK_READ_TIMER ( TIMER_NUM := timer_1,
                                          PT_TIME := ADDRESS (count));
            count := count_init - count;
            WRITELN (count,' pulses were counted');
          END;
END.
```

Figure 6–4 shows the appropriately modified YK driver prefix file. (The YK prefix file is described in Section 6.8.5.)

Figure 6–4: YK Prefix File for External Pulse Counter Sample Program

```
.TITLE  YKPFXT - Parallel I/O and counter/timer Driver Prefix Module
        .ident  /V2.0/
;+
;DEFINE PRIORITIES FOR YK HANDLER
;-
YK$HPR  == 5            ; Hardware priority
YK$IPR  == 250.         ; Initialization priority
YK$PPR  == 180.         ; Process priority

;+
;CALL:  INITIALIZE MACRO
;-
        .MCALL  YKCI$
        YKCI$
;+
; PORT A not used
;-
;+
; PORT B
; Bit port, Timer 1's gate input via bit 7 and count pulse input via bit 5
; (actually all 8 bits set up as inputs)
;-
        YKCP$  CHAN=B,PTYPE=YK$BIT

;+
; PORT C
; Bit port, Timer 3's output via bit 0
;-
        YKCP$  CHAN=C,PTYPE=YK$BIT,OUT=YK$BO
;+
; TIMER 1
; Enable timer 1's gate input and count input
;-
        YKCT$  TNUM=1,TEXTG=YES,TEXTC=YES
;+
; TIMER 2
; Not used
;-

;+
; TIMER 3
; Enable timer 3's output in one-shot mode
;-
        YKCT$  TNUM=3,TEXTO=YES,TOUT=YK$T1S
;+
; END CONFIGURATION
;-
        YKCE$

        .END
```

6.4.2.9 Linking Two Timer/Counters as 32-Bit Counter

Figure 6–5 shows a sample program, usable with the distributed YK driver prefix file, that links KXT11–CA/KXJ11–CA timers 1 and 2 together as a 32-bit counter.

With the values supplied—0 (66536) in timer 1 and 200 in timer 2—timer 1 times out every .033 seconds, and timer 2 times out every 6.6 (200*.033) seconds.

Figure 6–5: KXT11–CA 32-Bit Counter Sample Program

```
[SYSTEM(MICROPOWER), PRIORITY(100),
  DATA_SPACE (3000), STACK_SIZE (800)] PROGRAM YKTIM6;

{$NOLIST }
%include 'ykinc.pas'
%include 'escode.pas'    { Status codes }
{$LIST }

CONST
  clear = ''(27)'[2J'(27)'[H';

VAR
  yk_io_stat_2,
  yk_io_stat_1     : unsigned;
  sem_2_bin,
  sem_1_bin        : SEMAPHORE_DESC;
  bin_worked_2,
  bin_worked_1     : BOOLEAN;

[INITIALIZE] procedure foist;
BEGIN
  bin_worked_2 := CREATE_BINARY_SEMAPHORE (DESC := sem_2_bin );
  bin_worked_1 := CREATE_BINARY_SEMAPHORE (DESC := sem_1_bin );
END;

procedure cls;

BEGIN
  write(clear);
END;

BEGIN
  cls;
  IF ((bin_worked_1) AND (bin_worked_2)) THEN
  BEGIN
{This timer will be decremented by 1 each time Timer 1 times out.  Therefore,
 it will count to 0 after 200*33 millisec = 6.6 seconds}
    yk_io_stat_2 := YK_SET_TIMER ( TIMER_NUM := timer_2, TIMER_VALUE :=200,
                    MODE := [ init_constant, trigger, contin_cycle ],
                     BIN_SEM := ADDRESS(sem_2_bin));
```

```
    IF (yk_io_stat_2 <> es$nor) THEN
        BEGIN
            WRITELN ( 'Error setting timer 2');
        END
    ELSE
        BEGIN
            WRITELN ( 'Timer 2 started up' );
        END;
{Maximum value - so it times out every 33 milliseconds}
    yk_io_stat_1 := YK_SET_TIMER ( TIMER_NUM := timer_1, TIMER_VALUE := 0,
                      MODE := [ init_constant, trigger, contin_cycle ],
                      BIN_SEM := ADDRESS(sem_1_bin));

    IF (yk_io_stat_1 <> es$nor) THEN
        BEGIN
            WRITELN ( 'Error setting timer 1');
        END
    ELSE
        BEGIN
            WRITELN ( 'TIMER 1 STARTED UP');
        END;
  while true do
    begin
        wait ( desc := sem_2_bin );
        writeln ('timer 2 signaled');
    end;
  END
  ELSE
    BEGIN
        writeln ( 'Semaphores were not created for timers');
    END;
END.
```

6.5 Request/Reply Packet Interface

The packet-level functions provided by the parallel line drivers are listed by symbolic and decimal function code as follows:

Code	Function
IF$RDP (0)	Read Physical
IF$RDL (1)	Read Logical (equivalent to Read Physical)
IF$WTP (3)	Write Physical
IF$WTL (4)	Write Logical (equivalent to Write Physical)
IF$SET (6)	Set Characteristics (DRV11–B)
IF$GET (7)	Get Characteristics
IF$ENA (8)	Enable (DRV11–J)
IF$DSA (9)	Disable (DRV11–J)

Code	Function
IF$YKP (8)	Set Pattern (KXT11-CA/KXJ11-CA PIO)
IF$YKR (9)	DMA Read (KXT11-CA/KXJ11-CA PIO)
IF$YKW (10)	DMA Write (KXT11-CA/KXJ11-CA PIO)
IF$YKE (11)	DMA Complete (KXT11-CA/KXJ11-CA PIO)
IF$YKS (12)	Set Timer (KXT11-CA/KXJ11-CA PIO)
IF$YKU (13)	Clear Timer (KXT11-CA/KXJ11-CA PIO)
IF$YKT (14)	Read Timer (KXT11-CA/KXJ11-CA PIO)

If a request is received for an Open (IF$LOK or IF$ENT), a Close (IF$CLS), or a Purge (IF$PRG), the driver returns an illegal function status code (ES$IFN), which the ACP (Open) or OTS (Close/Purge) interprets as indicating that no device-dependent processing was required for that operation.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the parallel line drivers are listed by symbolic code and bit position as follows. All but the last are specific to the KXT11-CA/KXJ11-CA PIO driver and to a single type of operation:

Code	Function
FM$YKA (bit 9)	AND pattern mode—all bits specified in pattern argument must match (YK Set Pattern)
FM$YKO (bit 10)	OR pattern mode—match with any bit specified in pattern argument (YK Set Pattern)
FM$YKW (bit 11)	Sets wait-for-pattern-match mode (YK Set Pattern)
FM$YKT (bit 8)	Trigger timer after setup (YK Set Timer)
FM$YKI (bit 9)	Initialize timer constant value (YK Set Timer)
FM$YKC (bit 11)	Continuous cycle—cause timer to signal a binary or counting semaphore and restart after each timeout (YK Set Timer)
FM$YKR (bit 8)	Reset pattern mode—reset a previously set pattern mode at the end of a read or a write (YK PIO port A or B read/write)
FM$BSM (bit 13)	Signal binary/counting semaphore

The DRV11-J (XA) and DRV11-B (YB) drivers each consist of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each configured controller. I/O requests intended for a particular

controller are sent (using a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The DRV11 (YA) driver consists of one static process and two dynamic processes—a read process and a write process. After completing its initialization functions at the initialization priority specified in the prefix file, the static process lowers its priority and functions as a dispatcher process for I/O requests. The dispatcher process performs request-handling functions for both dynamic processes.

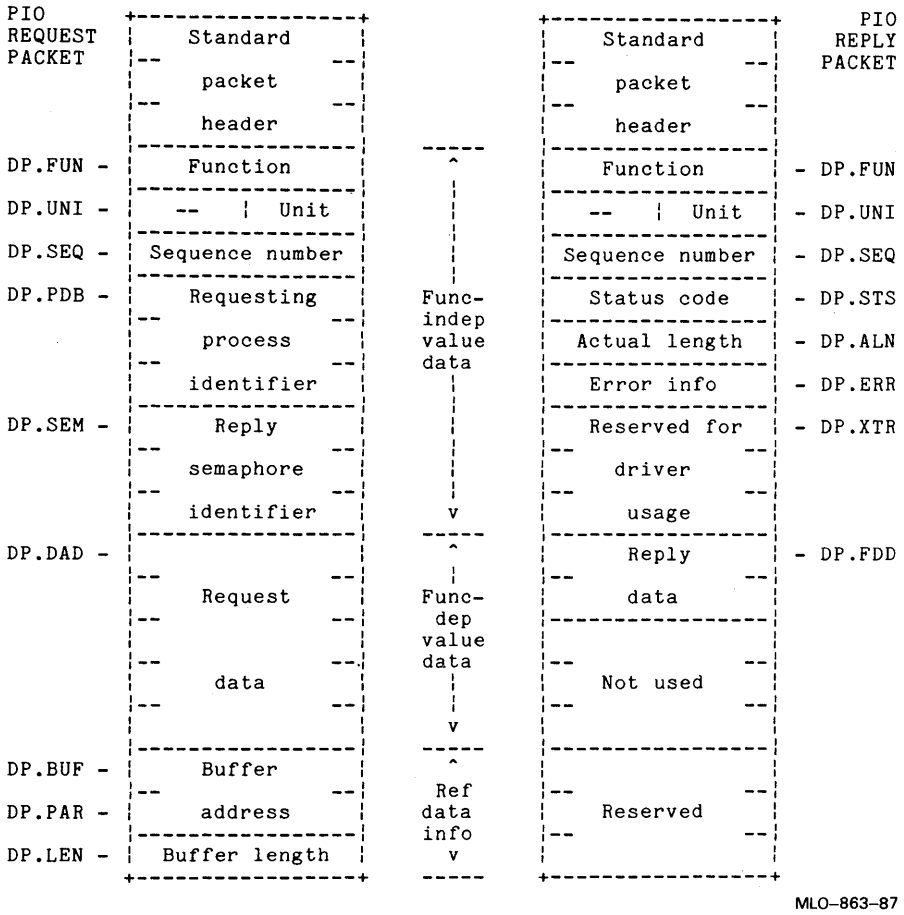
The SBC-11/21 PIO (YF) and KXT11-CA/KXJ11-CA PIO (YK) drivers each consist of a single (static) process. After completing its initialization functions at the initialization priority specified in a prefix file, each static process lowers its priority and functions as a dispatcher process for I/O requests.

The following list shows the request queue names and number of supported units for parallel line driver requests:

Driver	Request Queue Name	Number of Units	Numbering
DRV11-J	$XAc	[For read/write:] 1-4	0 through 3 for ports A through D
		[For Enable/Disable:] 1-12	4 through 15 for port A lines 0 through 11
DRV11	$YAA	1	0
DRV11-B	$YBc	1	0
SBC-11/21 PIO	$YFA	1-2	0 and 1 for ports A and B
KXT11-CA or KXJ11-CA PIO	$YKA	1-6	0 through 2 for ports A through C and 3 through 5 for timers 1 through 3

The letter c in a queue name represents a controller designation (A, B, ..., as specified in a parallel line driver prefix file). The number of units actually configured for each controller and their unit numbers must be specified in a driver prefix file.

The general format of the parallel I/O request and reply packets is shown below:



The function-independent portions of the previous packets are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of parallel line driver function.

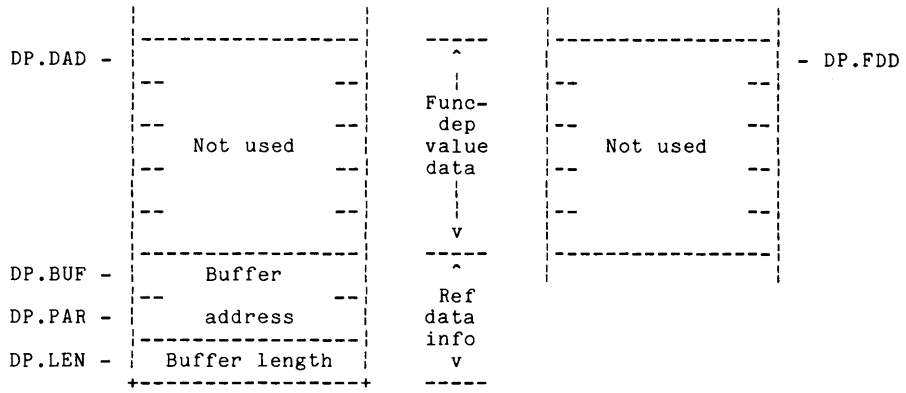
Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

6.5.1 DRV11-J (XA) Functions

6.5.1.1 XA Read and Write

The XA read and write functions transfer an even number of bytes to or from a user-specified buffer. The function-dependent portions of the XA read/write request and reply packets are shown below:

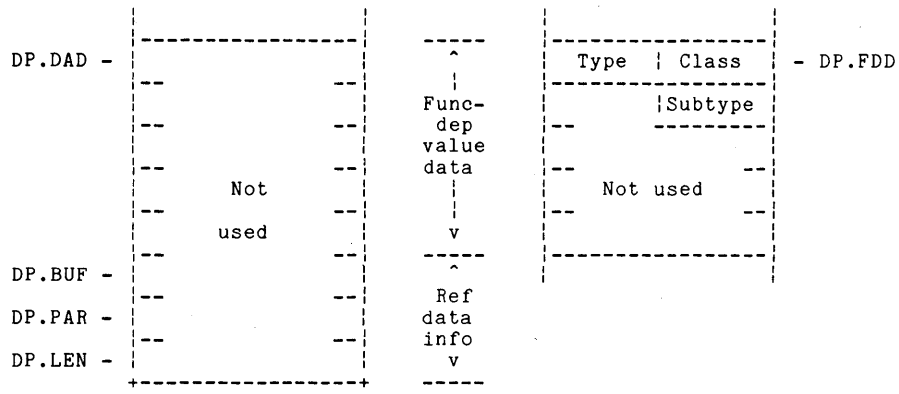


The buffer address, which must be on a word boundary, specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer. If the length value is an odd number, the last byte is not transferred.

6.5.1.2 XA Get Characteristics

The Get Characteristics function returns a block of device-dependent information. That information consists of device class, type, and subtype.

The function-dependent portions of the XA Get Characteristics request and reply packets are shown below:



MLO-865-87

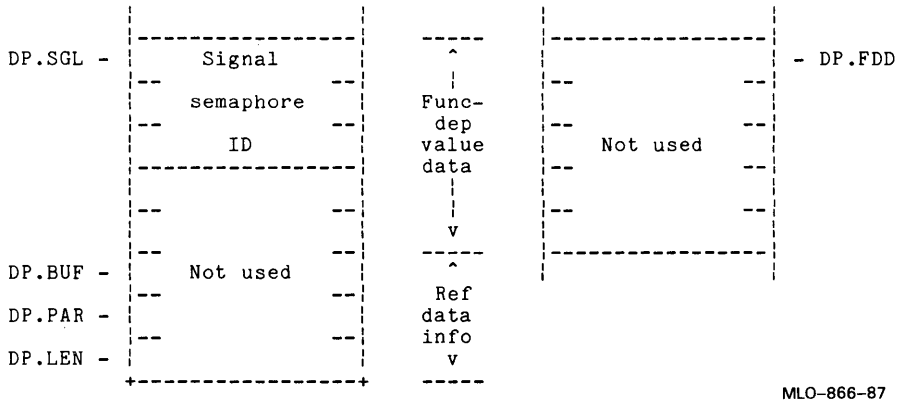
In the information above:

- Class is DC$RLT for real-time device class.
- Type is RT$DRJ for DRV11-J device type.
- Subtype is dependent on the unit number for XA:
  RS$SLN = sense-line subtype (units 4 through 15)
  RS$PRT = parallel-port subtype (units 0 through 3)

6.5.1.3 XA Enable

The XA Enable function enables interrupts on a specified interrupt line (unit 4 to 15) of port A, if the line is masked, and associates the user's signal semaphore with that line. The signal semaphore is a binary or counting semaphore, distinct from the user's reply semaphore. The semaphore is signaled each time the corresponding interrupt occurs (line goes active-low). If the specified interrupt line is already enabled when the request is issued, the Enable function returns a "line already attached" (ES$DAL) error status. (A Disable request must intervene between two Enable requests for a given line.)

The function-dependent portions of the XA Enable request and reply packets are shown below:



Field DP.SGL specifies, by structure ID, the user's semaphore that is to be attached to the selected interrupt line. This semaphore is signaled each time an interrupt occurs until that line is detached. The unit number selects the desired bit-interrupt line; the range of valid unit numbers is 4 to 15 for port A lines 0 to 11, respectively.

6.5.1.4 XA Disable

The XA Disable function masks interrupts on a specified interrupt line (unit 4 to 15), severing the association between that line and a signal semaphore, if any. A Disable request for a masked line—never enabled or previously disabled—has no effect and returns a normal status.

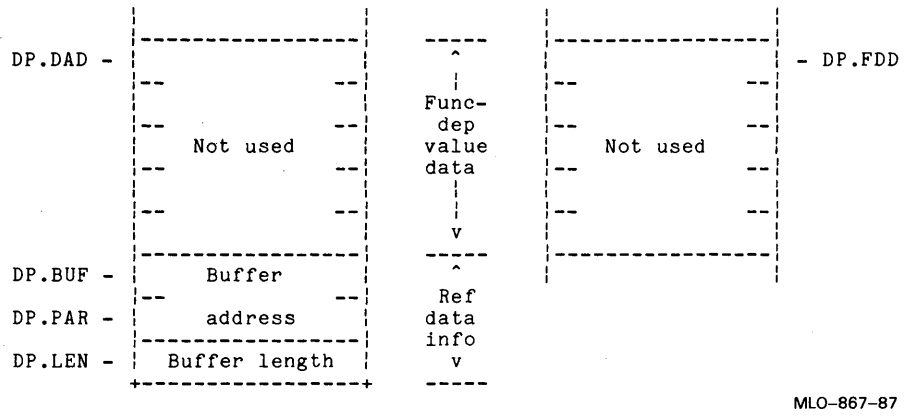
The function-dependent portions of the XA Disable request and reply packets are not used.

6.5.2 DRV11 (YA) Functions

6.5.2.1 YA Read and Write

The YA device driver supports simultaneous input and output on a single unit. The driver assumes that the REQ A signal represents an output interrupt request and that the REQ B signal represents an input interrupt request; the driver enables the corresponding DRCSR status bits for those purposes. Read and write requests transfer an even number of bytes to or from a user-specified buffer.

The function-dependent portions of the YA read/write request and reply packets are shown below:



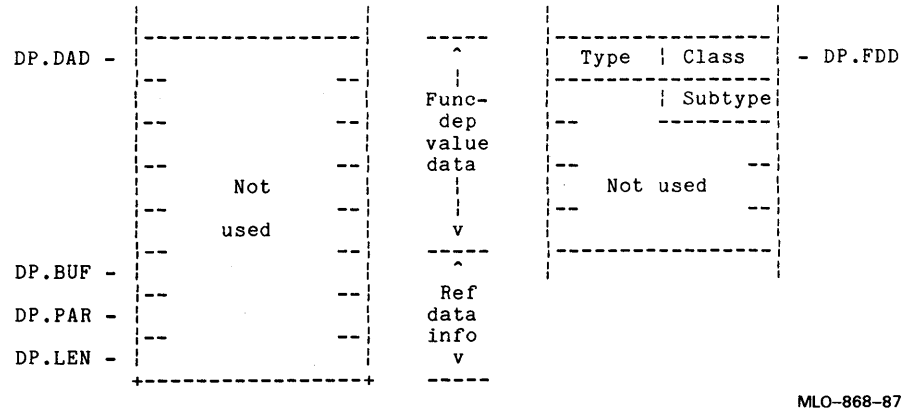
MLO-867-87

The buffer address, which must be on a word boundary, specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer. If the length value is an odd number, the last byte is not transferred.

6.5.2.2 YA Get Characteristics

The Get Characteristics function returns a block of device-dependent information. That information consists of device class, type, and subtype.

The function-dependent portions of the YA Get Characteristics request and reply packets are shown below:



MLO-868-87

In the previous information:

- Class is DC$RLT for real-time device class.
- Type is RT$DRV for DRV11 device type.
- Subtype is RS$PRT (parallel-port subtype).

6.5.3 DRV11-B (YB) Functions

6.5.3.1 YB Read and Write

The YB driver supports transfers of up to 32K 16-bit words per request in single-cycle or burst mode, with 18-bit buffer addressing.

The direction of DMA transfer (user device to memory or vice versa) cannot be directly set under program control. Instead, the transfer direction is set by the C0 and C1 TTL input lines from the user device to the DRV11-B, as follows:

Transfer Type	C0	C1
Memory to user device (DATI)	0	0
Memory to user device and back (DATIO)	1	0
User device to memory (DATO)	0	1
User device to memory (DATOB)	1	1

The program controls the setting of the C0 and C1 lines by manipulating the user device via the FNCT 1, FNCT 2, and FNCT 3 control lines. The user device must interpret the FNCT control line settings and then set the appropriate C0 and C1 combination.

Read or write requests transfer an even number of bytes to or from a user-specified buffer.

Upon receiving a read or write request, the driver validates the request, sets up the Bus Address Register (BAR) and the Word Count Register (WCR) to initialize the DMA transfer, and sets the CSR, based on an internally maintained default CSR setting. The default settings must have been previously set by YB Set Characteristics commands. Setting the CSR initiates a DMA transfer either immediately (GO and CYCL bits set) or when the device indicates its readiness by asserting CYCLE REQUEST for at least 1 microsecond (GO bit set and CYCL bit cleared). When multiple DMA requests are posted to the driver, the requests are validated and queued internally to minimize the latency in switching from the completion of one request to initiating the next request.

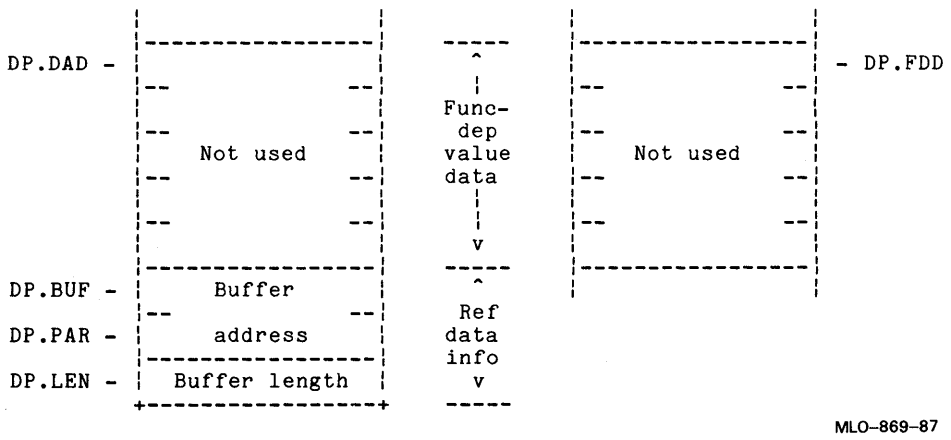
Under normal circumstances, the maximum length for the user source or destination buffer is 8128 (decimal) bytes. This ensures that the user's buffer can be mapped using a single driver process PAR. However, larger buffers are supported, provided they occupy contiguous physical memory. The limit for buffer size is 64K bytes, which is the limit imposed by the DRV11-B hardware.

For buffers larger than 8128 (decimal) bytes, perform the following steps to ensure that DMA transfers will take place without accidentally corrupting sections of memory that are being used for other purposes:

1. Reserve a contiguous block of physical memory for the buffer or buffers. (Multiple buffers may be required if a second or subsequent DMA transfer in the same direction must be initiated before the first buffer has been saved or reloaded.) The amount of physical memory required per buffer is the sum of the whole number of 64-byte blocks required to contain the buffer. The required memory may be reserved for buffer use by NOT specifying it in the system configuration file during the application build.
2. To enable DMA transfers of the required length between the user device and LSI-11 bus (Q-bus) memory, the user's process must directly write the appropriate value data and reference-data specification—including an unconfigured-memory PAR value—into the request packet, and set the "reference-data-present" packet header bit (bit 15 of the control/priority word). To signal the appropriate YB request queue semaphore, PUT_PACKET or SGLQ$ must be used instead of SEND or SEND$ (which supplies configured-memory PAR values). The user-supplied PAR value will be used by the driver process to map to the base of the user buffer before initiating transfers.
3. When the DMA transfers are complete, the user process must map directly to the buffer to recover/reload the buffer contents page by page.

The DRV11-B module is capable of performing DMA in single-cycle mode as well as burst mode. The default is single-cycle mode. Burst mode cannot be requested directly by the user's program. Instead, the user device must be placed in a mode where it requests a burst mode cycle from the DRV11-B (by holding SINGLE CYCLE low). The user device can be manipulated under program control to request single-cycle or burst mode transfers as desired, by using the FNCT 1, FNCT 2, and FNCT 3 control lines. This assumes that the user device has the necessary intelligence to interpret those control signals.

The function-dependent portions of the YB read or write request and reply packets are shown below:



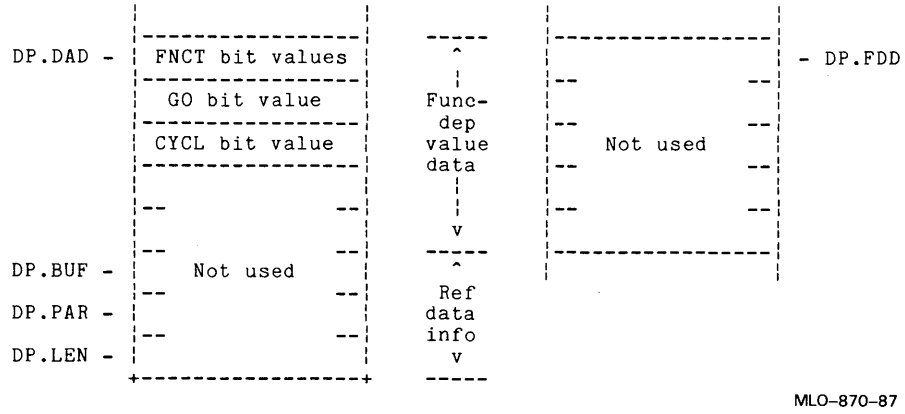
The buffer address, which must be on a word boundary, specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer. If the length value is an odd number, the last byte is not transferred.

6.5.3.2 YB Set Characteristics

The YB Set Characteristics function establishes internal default CSR settings to be used for subsequent DMA transfers. The settable bits are the FNCT bits, available for user-defined purposes; the GO bit, which indicates readiness for a DMA transfer; and the CYCL bit, which determines whether the transfer is initiated by the LSI-11 bus (Q-bus) arbiter or the user device. Initial default settings of those bits are determined in the YB driver prefix file.

Upon receiving a Set Characteristics request, the DRV11-B driver clears the CSR, resets the three FNCT bits according to passed bit values, and resets the default settings of the FNCT, GO, MAINT, and CYCL bits. The new settings remain in effect until they are explicitly reset. (Note that only the FNCT bit settings are moved immediately into the CSR.)

The function-dependent portions of the YB Set Characteristics request and reply packets are shown below:



The three CSR value fields shown above correspond to the YB prefix file symbols YB$FNC, YB$GO, and YB$CYC.

The FNCT values field (offset DP.DAD) supplies the default values for the CSR bits FNCT 1 (bit 1), FNCT 2 (bit 2), and FNCT 3 (bit 3). The allowable values for the field reflect the bit positions—even decimal values 0 through 14 (binary 1110) inclusive. The FNCT bits are available to the user device for user-defined purposes.

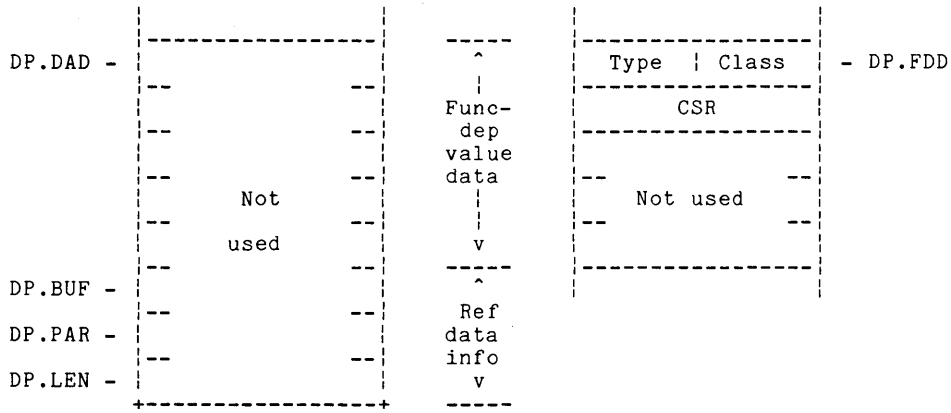
The GO value field (offset DP.DAD+2) supplies the default value for the CSR bit GO (bit 0). It may be 0 or 1. Setting the GO bit indicates that the DRV11-B registers have been set up for a DMA transfer, which can then be initiated by either the user's program or the user device.

The CYCL value field (offset DP.DAD+4) supplies the default value for the CSR bit CYCL (bit 8). It may be 0 or 1. The CYCL bit indicates whether DMA transfers are to be initiated by the Q-bus arbiter (bit set) or by the user device (bit clear). Setting the CYCL bit with a Set Characteristics command causes a DMA transfer to be initiated as soon as the next transfer request is posted to the driver process. Clearing the bit with Set Characteristics means that the actual DMA transfer must be initiated by the user device after a transfer request has been posted by the application program. The CYCL bit must be set when the DRV11-B is being used in maintenance mode.

6.5.3.3 YB Get Characteristics

The Get Characteristics function returns a block of device-dependent information. That information consists of device class and type and the current contents of the device CSR.

The function-dependent portions of the YB Get Characteristics request and reply packets are shown below:



MLO-871-87

In the information above:

- Class is DC$RLT for real-time device class.
- Type is RT$DRB for DRV11-B device type.
- CSR is a copy of the current Control and Status register, from which the user can extract STAT A, STAT B, and STAT C.

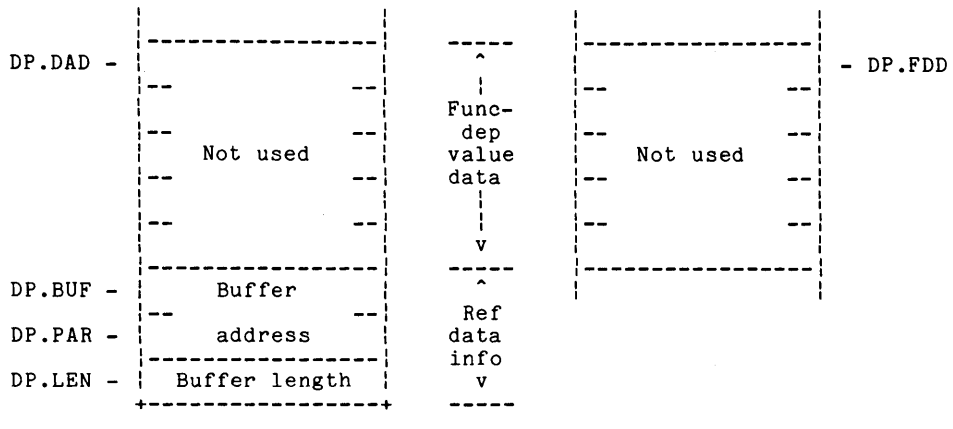
6.5.4 SBC-11/21 PIO (YF) Functions

6.5.4.1 YF Read and Write

The YF driver supports unidirectional input and/or output with 8-bit transfers as selected by jumpers on the module. The driver supports only mode 1 PIO transfers; modes 0 and 2 are not supported. Mode 1 uses the standard (factory) hardware configuration and permits interrupt-driven transfers in block mode.

Read and write requests transfer a specified number of bytes to or from a specified buffer.

The function-dependent portions of the YF read or write request and reply packets are shown below:



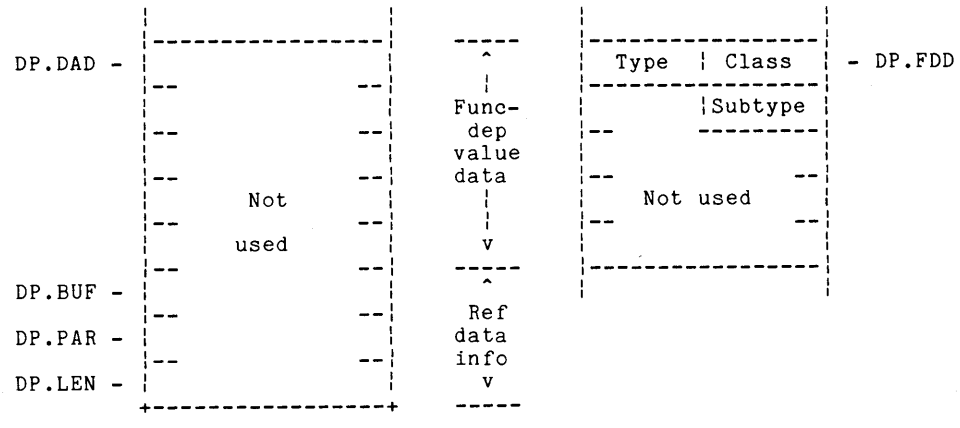
MLO-872-87

The buffer address, which must be on a word boundary, specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer. If the length value is an odd number, the last byte is not transferred.

6.5.4.2 YF Get Characteristics

The Get Characteristics function returns a block of device-dependent information. That information consists of device class, type, and subtype.

The function-dependent portions of the YF Get Characteristics request and reply packets are shown below:



MLO-873-87

In the previous information:

- Class is DC$RLT for real-time device class.
- Type is RT$FAL for SBC-11/21 PIO type.
- Subtype is RT$PRT for parallel-port subtype.

6.5.5 KXT11-CA/KXJ11-CA PIO (YK) Functions

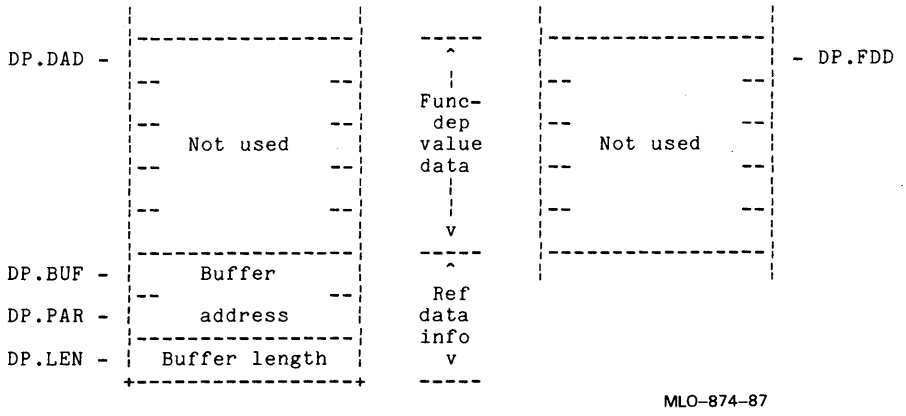
6.5.5.1 YK Read

The YK read functions transfer data from a parallel port to a KXT11-CA/KXJ11-CA buffer.

YK read operations are affected by the function modifiers specified in Set Pattern operations. In pattern-match mode, read operations terminate either when a user-specified pattern is found or when a user-specified search limit (DP.LEN) expires. If no pattern modifiers were specified in a YK Set Pattern request, the default is FM$YKA (all bits must match).

If pattern-match mode is not set, you specify the length of the transfer.

For bit ports, a read request must be for one byte for a single port or two bytes for linked ports. The function-dependent portions of the YK read request and reply packets are shown below:



MLO-874-87

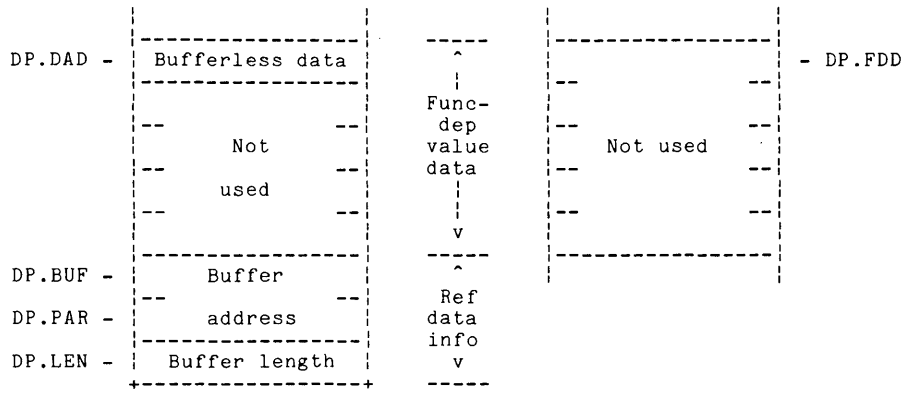
6.5.5.2 YK Write

The YK write functions transfer data from a KXT11-CA or KXJ11-CA buffer to a parallel port.

YK write operations, like read operations, are affected by the function modifiers specified in Set Pattern operations. In pattern-match mode, write operations terminate either when a user-specified pattern is found or when a user-specified search limit (DP.LEN) expires. If no pattern modifiers were specified in a YK Set Pattern request, the default is FM$YKA (all bits must match).

If pattern-match mode is not set, you specify the length of the transfer.

For bit ports, a write request must be one byte for a single port, or two bytes for linked ports. The function-dependent portions of the YK write request and reply packets are shown below:



MLO-875-87

If you are writing a single byte to a port or two bytes to a linked port, you can omit the buffer specification and place the data at offset DP.DAD. You must, however, send by value only (and not by value and by reference).

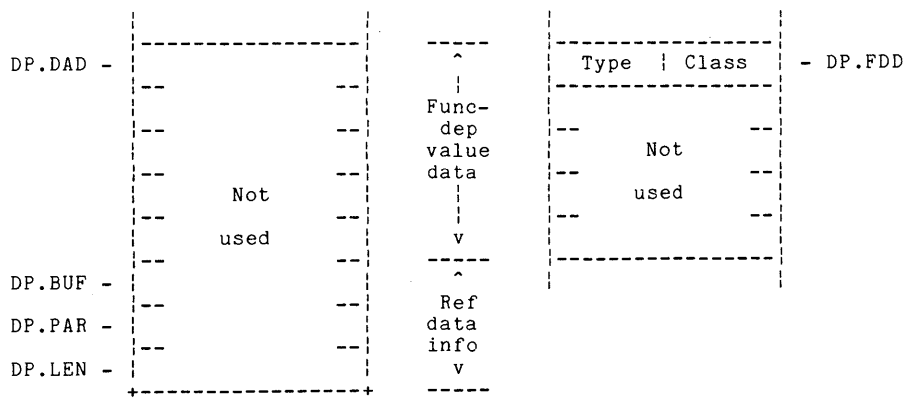
Note

The support routine YK_PORT_WRITE does not support this technique.

6.5.5.3 YK Get Characteristics

The YK Get Characteristics function returns a block of device-dependent information, which consists of device class and type.

The function-dependent portions of the YK Get Characteristics request and reply packets are shown below:



MLO-876-87

In the previous information:

- Class is DC$RLT for real-time device class.
- Type is RT$YKP for KXT11-CA or KXJ11-CA parallel port or RT$YKT for KXT11-CA or KXJ11-CA timer.

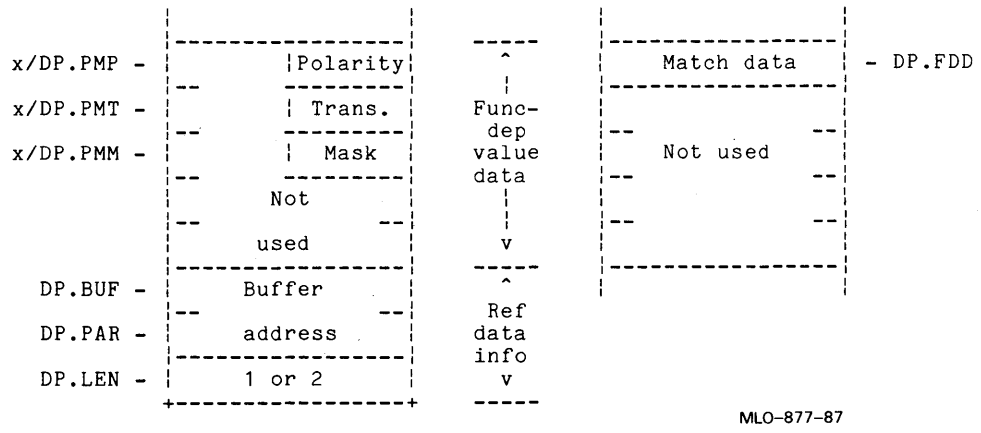
6.5.5.4 YK Set Pattern

The YK Set Pattern function sets pattern-match mode on parallel port A or B. This allows you to terminate a data transfer when a user-specified pattern is found.

The function modifier bits for this request specify whether all bits specified in the pattern argument must match (FM$YKA), or just one bit in the pattern argument must match (FM$YKO), and whether wait-for-pattern-match mode is to be set (FM$YKW). If no modifiers are specified, the default is FM$YKA.

To use pattern matching, you must specify "PAT=YES" for port A or B in the prefix file port-configuration (YKCP$) macro.

The function-dependent portions of the YK Set Pattern request and reply packets are shown below:



Fields DP.PMP, DP.PMT, and DP.PMM collectively define the match pattern for the specified port (unit). Each bit in a DP.PMP, DP.PMT, and DP.PMM field corresponds to a bit (0–7) in the match pattern; that is, each bit of the match pattern is defined by corresponding bits of DP.PMP, DP.PMT, and DP.PMM. For each match pattern bit, DP.PMP supplies pattern-polarity information; DP.PMT, pattern-transition information; and DP.PMM, pattern-mask information.

The pattern specification for each bit of the match pattern is defined as follows:

DP.PMP	DP.PMT	DP.PMM	Event Recognized
x	0	0	Bit masked off—no event recognized
x	1	0	Any transition
0	0	1	Logical 0 state
1	0	1	Logical 1 state
0	1	1	Logical 1 to logical 0 transition
1	1	1	Logical 0 to logical 1 transition

Note

Do not specify more than one bit to detect transitions if you specify AND pattern mode (function modifier FM$YKA).

For example, to set a pattern of bits 0 to 3 = 1 AND bits 5 and 6 = 0 AND bit 7 = logical 1 to logical 0 transition AND bit 4 ignored, you would specify AND pattern mode and pass the bit pattern 00001111 in DP.PMP, 10000000 in DP.PMT, and 11101111 in DP.PMM. For wait-for-pattern-match mode, you can use the DP.BUF, DP.PAR, and DP.LEN fields to describe a variable for receiving the matching data. The length field should be 1 for a single port and 2 for linked ports. Alternatively, if these fields are left clear and the packet is sent by value only, then the matching data is returned in the first word of the function-dependent portion of the packet, as shown above.

6.5.5.5 YK DMA Read, Write, and Complete

The YK PIO DMA functions allow you to perform DMA transfers via a KXT11-CA or KXJ11-CA parallel port. To perform a DMA transfer, you first send a DMA Read or DMA Write request to the YK driver and wait for a reply. If the reply indicates normal status, you then send a DMA transfer command to the DMA (QD) driver and wait for the request to complete. After the DMA transfer completes, you send a DMA Complete request to the YK driver. The DMA Complete request unlocks the request queue for the port that was used for the transfer.

The function-dependent portions of the YK DMA request and reply packets are not used.

For guidelines to follow when performing DMA I/O on a KXT11-CA or KXJ11-CA parallel port—and a sample Pascal program—see Section 6.4.2.4.

6.5.5.6 YK Set Timer

The YK timer functions control the timer/counters on the KXT11–CA and KXJ11–CA. Depending on the function modifier specified, the Set Timer function can initialize a timer constant (FM$YKI), trigger the timer after setup (FM$YKT), or set up a timer to periodically signal (continuous cycle mode) a binary or counting semaphore (FM$YKC).

If none of the three YK Set Timer modifiers is specified, the timer mode set by the prefix file or by the last timer command remains in effect.

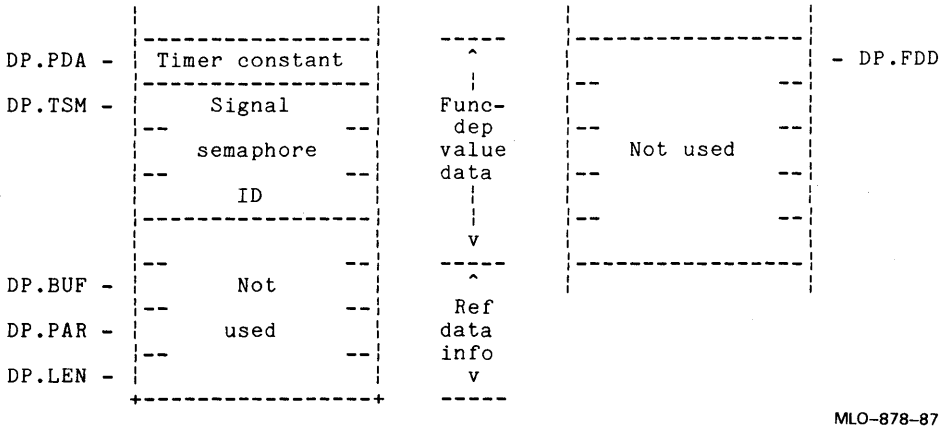
If periodic signaling is specified, the driver signals a binary or counting semaphore and restarts the timer after each timeout. In such a case, the driver replies to the user right after the timer is set up. If periodic signaling is not specified, the timer just counts down once (single cycle), and the driver then replies to the user.

Timers are initialized to a value (offset DP.PDA, below) and count down to 0. You subtract the current value from the initial value to calculate the number of ticks.

The timers count down at a rate of 2MHz, or one tick every .5 microseconds. At that rate, counting down to 0 from the maximum 16-bit timer value (65536) takes approximately 33 milliseconds. For longer intervals, you can have the timer count from 65536 to 0 several times or link two timers together to make a 32-bit timer. To link two timers, use "TLNK=YK$1I2" in the YK prefix file macro YKCT$.

Section 6.4.2.8 provides a sample program that uses the YK support routines and timers 1 and 3 to count external pulses. Section 6.4.2.9 provides a sample program that uses the YK support routines to manipulate a 32-bit timer—timers 1 and 2 linked together.

The function-dependent portions of the YK Set Timer request and reply packets are shown below:



MLO-878-87

Field DP.PDA specifies a timer constant value to be set. You must provide a signal semaphore ID in field DP.TSM if you want to establish a continuous-cycle semaphore to be signaled upon each timer timeout. In addition, you must send the packet by value only.

Note

If port C is being used to supply handshake signals while timer 3 is being used as a general-purpose timer, the time constant must be set for timer 3 during initialization and not changed during operation. The reason is that port C is disabled during the setting of timer 3's constant, and therefore the handshake signals also get disabled.

6.5.5.7 YK Clear Timer

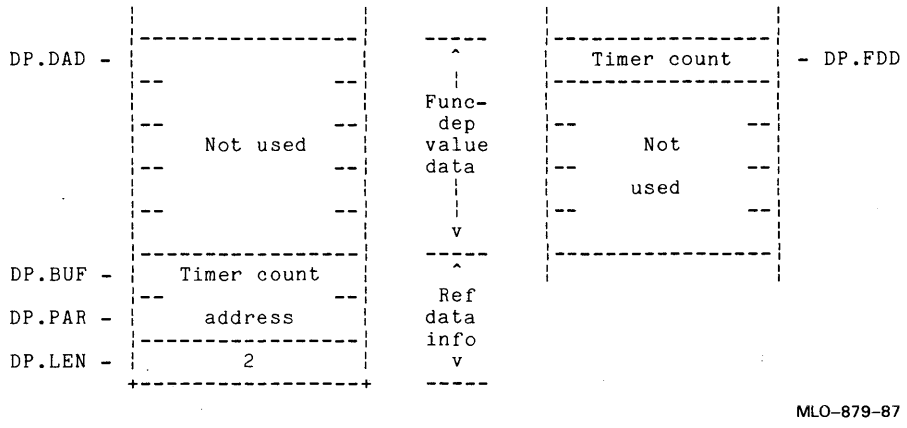
The YK Clear Timer function deactivates a timer that is operating in the continuous mode.

The function-dependent portions of the YK Clear Timer request and reply packets are not used.

6.5.5.8 YK Read Timer

The YK Read Timer function returns the current count value from a timer.

The function-dependent portions of the YK Read Timer request and reply packets are shown below:



MLO-879-87

The DP.BUF, DP.PAR, and DP.LEN fields contain information about the variable that is to receive the timer-count value. If these fields are left clear, and the packet is sent by value only, the timer-count value will be returned in the first word of the function-dependent portion of the reply packet, as shown above.

6.6 Status Codes

If an error is detected during an I/O operation by a parallel interface or driver, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified).

If no error was detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code field of the reply message.

The parallel line drivers return the following exception codes in the status field of the reply message:

Code	Type	Description
ES$ABT	HARD_IO	I/O abort, driver process deleted, request not serviced (XA, YB, YF)
ES$ATN	HARD_IO	Device has indicated error by asserting ATTN bit in the CSR (YB—see Section 6.7, Extended Error Information)
ES$DAL	HARD_IO	Device allocated (XA); line already attached (XA Enable)
ES$IVM	HARD_IO	Invalid mode: DMA transfers are not allowed on bit type ports, PIO cannot use DMA when request line output is defined as an input, pattern match not allowed on PIO port B when linked to port A, transition recognition cannot be used on PIO port using a handshake, if DMA transfer is used on one PIO port, then other port must be bit type (YK)
ES$IVP	HARD_IO	Invalid request packet parameter: odd number of bytes to transfer (YB)
ES$NXM	HARD_IO	Nonexistent or read-only memory (YB)
ES$NXU	HARD_IO	Nonexistent unit (XA, YB, YF, YK)
ES$OVF	HARD_IO	Buffer size was exceeded before a PIO pattern match was detected (YK)
ES$PWR	HARD_IO	Power failure (YB)
ES$IFN	SOFT_IO	Invalid function code (XA, YA, YB, YK); attempt to perform a supported function on a PIO unit not configured to support that function (YF); also used internally to signal ACP or OTS that no device-dependent processing of an Open, Close, or Purge was required

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO–11 users.

Note

Not listed above are exception codes for OTS-detected I/O errors or for kernel-detected errors that the parallel drivers raise rather than passing back to the requesting process. OTS-detected I/O errors are listed in Chapter 9 of the *MicroPower/Pascal Language Guide*.

6.7 Extended Error Information

In the event of a DRV11–B hardware error (ATTN bit asserted in the CSR), the YB driver copies the CSR into the DP.ERR field of the driver reply packet. The CSR is described in the DRV11–B hardware guide.

6.8 Parallel Line Driver Prefix Files

Figures 6–6 through 6–10 show the parallel line driver prefix modules. The following sections describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

6.8.1 XA Prefix File

The XA driver prefix file (Figure 6–6) specifies that low-active signals will be used for generating interrupt requests. If high-active signals will be used, edit XAPFX.MAC to reflect that configuration.

The driver prefix file specifies the use of rotating priority for interrupts within each of two 8-bit groups. When rotating priorities are used, group 1 consists of port A I/O bits 0 to 7; group 2 consists of port A I/O bits 7 to 11 and USER RPLY A to D (standard hardware configuration), or port A I/O bits 7 to 15 (nonstandard hardware configuration). Rotating priority assigns the lowest interrupt priority within a group to the line that most recently received interrupt service. Thus, a maximum of eight interrupt cycles would be required for that group to service each interrupt request. Group 1 always has higher interrupt priority than group 2.

When fixed priority is used, edit XAPFX.MAC to reflect that hardware configuration. Fixed priority (nonrotating) within a group causes interrupt priority to be determined by the physical position of each line on the I/O connector. The highest-priority line is port A I/O line 0; the lowest-priority line is the USER RPLY D line. However, when USER RPLY interrupts are disabled through the use of the nonstandard hardware configuration (W11 removed), the lowest-priority line is port A line 15. When an application requires fixed interrupt priority for each line within a group, the prefix file can be modified to meet this requirement. Change the value of the symbol J$RPRI from 1 to 0, as indicated in the source comments, and reassemble the prefix file before installing it in the system.

The DRVCF$ macro specifies the driver name for the request-queue semaphore and the number of DRV11–J devices (controllers) on the target to be supported by the driver. You can edit that field to change the number of controllers; if you do, however, you must add a CTRCF$ macro for each controller.

The CTRCF$ macro specifies the controller name, number of units—ports and bit-interrupt lines—the controller supports, CSR and vector addresses, and the unit numbers of supported units.

Change cname = A to cname = B for the second controller, cname = C for the third, and so on. The cname field supplies the third character for the corresponding request-queue semaphore name.

When specifying nunits, be sure to include the decimal point. If the DRV11–J device on your target supports fewer than 16 units—the maximum is 16—edit that field.

The csrvec field specifies both the initial CSR address (CSRA) and the first of 16 consecutive vector addresses associated with the device. You can edit that field if your device is not configured for those starting addresses.

You must use the DEVICES macro to specify all 16 interrupt vectors in the system configuration file. If the first vector is 400, for example, you must specify 400, 404, 410, 414, 420, 424, 430, 434, 440, 444, 450, 454, 460, 464, 470, and 474.

The units field specifies the unit numbers of the ports and sense lines supported by the controller. The designation 0:15 specifies units 0 through 15. Each unit number has a fixed significance, as defined by the XA driver. Unit numbers 0 through 3 refer to parallel ports A, B, C, and D of the DRV11-J, respectively. Unit numbers 4 through 15 refer to the individual bit-interrupt (sense) lines 0 through 11 of port A, respectively. The driver uses the values specified in the units field to validate the unit numbers given at run time in XA-driver function requests. You can edit that field to restrict the range of valid unit numbers for your device configuration. For example, the units specification units= <1,2,4:9> indicates that only ports B and C (units 1 and 2) can be referred to in parallel-I/O read or write requests and that only the sense lines 0 through 5 of port A (units 4 through 9) can be referred to in bit-interrupt Enable and Disable requests.

The symbols XA$xPR define various priorities associated with the driver, including the device interrupt priority.

The J$RPRI definition sets rotating priorities on or off—1 for rotating priorities, 0 for fixed priorities. J$HIGH sets interrupt polarity to high or low—0 for low-level polarity, 1 for high-level polarity.

Figure 6-6: DRV11-J Driver Prefix File (XAPFX.MAC)

```
        .title  XAPFX  - DRV-11J Device driver prefix module
;
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1982, 1986 by Digital Equipment Corporation.
; All rights reserved.
;

        .mcall  drvcf$
        .mcall  ctrcf$

XA$PPR  ==  175.                ; Process priority
XA$HPR  ==  4                   ; DRV-11J hardware priority
XA$IPR  ==  250.                ; Process initialization priority

J$RPRI  ==  1                   ; Set for rotating priorities
J$HIGH  ==  0                   ; Clear for low level interrupt polarity

        drvcf$  dname=XA,nctrl=1
        ctrcf$  cname=A,nunits=16.,csrvec=<164160,400>,units=<0:15>

        .end
```

6.8.2 YA Prefix File

In Figure 6-7, the YA STD_PROC_PRIO constant sets the process software priority to the standard value for the YA driver. The STD_INT_PRIO constant sets the default DRV11 device interrupt priority.

The CSR address is assigned the default value 167770. That is the factory-configured CSR address for the device.

The VECTOR_REQA and VECTOR_REQB assignments set the output (REQ A) interrupt vector address to 340 and the input (REQ B) interrupt vector address to 344, respectively. Those are the factory-configured vector addresses for the device.

Figure 6–7: DRV11 Driver Prefix File (YAPFX.PAS)

```
MODULE YAPFX;
{
 YAPFX.PAS  - Edit Level 1

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

 Copyright (c) 1982, 1986 by DIGITAL EQUIPMENT CORPORATION.
 All rights reserved.
}

const
   std_proc_prio = 175;          { Standard process priority }
   std_int_prio = 4;             { and interrupt priority }

var
   stdprio : [global] integer;              { Driver process priority }
   drprio : [global] integer;               { Interrupt priority }
   csr : [global] unsigned;                 { CSR address }
   vector_reqa : [global(REQA)] unsigned;   {vector address, A request}
   vector_reqb : [global(REQB)] unsigned;   {vector address, B request}
   maintenance : [global(MAINT)] boolean;

   { Declared in YADRV module }
   version : [external($YAVER)] packed array[1..6] of char;

[global] procedure initcsr;

begin
   stdprio := std_proc_prio;     { Final process priority }
   drprio :=  std_int_prio * 32; { In PSW placement }
   csr := %O'167770';            { CSR }

{
 Note that the interrupt vector addresses below are not the default
  addresses specified for the DRV-11.  The default vector addresses
  are 300 and 304.  The default addresses conflict with some other
  devices more commonly used.  Change the following addresses to 300
  and 304 if your hardware configuration requires it.
 }
   vector_reqa := %o'340';
   vector_reqb := %o'344';
   version := 'V02.00';
   maintenance := false;                {Not maintenance mode}
end   { procedure initcsr };
```

6.8.3 YB Prefix File

The YB prefix file assigns hardware and driver process priorities, assigns default settings for DRV11–B CSR bits, and invokes the DRVCF$ and CTRCF$ macros. (See Figure 6–8.)

The symbols YB$FNC, YB$GO, YB$MAN, and YB$CYC determine the default settings for device CSR bits. You can set alternate defaults by editing those global symbol definitions. You can also reset the CSR bits at run time via Set Characteristics requests from your application program.

YB$FNC supplies the default values for the CSR bits FNCT 1 (bit 1), FNCT 2 (bit 2), and FNCT 3 (bit 3). The allowable values for the symbol reflect the bit positions—even values 0. through 14. (binary 1110) inclusive. The FNCT bits are available to the user device for user-defined purposes.

YB$GO supplies the default value for the CSR bit GO (bit 0), which may be 0 or 1. Setting the GO bit indicates that the DRV11–B registers have been set up for a DMA transfer, which can then be initiated by either the user's program or the user device.

YB$MAN supplies the default value for the CSR bit MAINT (bit 12). It may be 0 or 1. The MAINT bit is set when the device is being tested in loopback mode.

YB$CYC supplies the default value for the CSR bit CYCL (bit 8). It may be 0 or 1. The CYCL bit indicates whether DMA transfers are to be initiated by the LSI–11 bus (Q-bus) arbiter (bit set) or by the user device (bit clear). Setting the CYCL bit at run time with a Set Characteristics command—see Section 6.5 (Request/Reply Packet Interface)—causes a DMA transfer to be initiated as soon as the next transfer request is posted to the driver process. Clearing the bit with Set Characteristics means that the actual DMA transfer must be initiated by the user device after a transfer request has been posted by the application program. YB$CYC must be set when the DRV11–B is being used in maintenance mode.

The DRVCF$ macro specifies the device name (YB) and the number of controllers to be supported by the driver.

The CTRCF$ macro is invoked once for each controller to be serviced by the driver. It specifies the controller identifier (A, B, ...), the number of units (always 1), CSR and vector addresses, and unit numbers.

6.8.4 YF Prefix File

The YF driver prefix file YFPFX.MAC (Figure 6–9) assumes the standard configuration for PIO ports. The standard configuration for PIO transfer direction is port A (Unit 0) for input and port B (Unit 1) for output. Port A can be configured for output transfers, and port B can be configured for input transfers, or both ports can be configured for input or output transfers, as required for a particular system application. If desired, only one port may be used. However, if any changes are made to the standard configuration, you must edit the YFPFX.MAC file to reflect those changes prior to building the application software.

YF$PPR defines the process software priority for the YF driver. The field can be modified to fine-tune the relationship between the driver and other processes in the application.

YF$FPR defines the driver's interrupt service routine fork priority. By default, the value of YF$FPR is set to 256 times the software priority.

YF$IPR defines the process initialization priority. That priority should be equal to that of all other I/O device drivers.

YF$HPR defines the hardware interrupt priority. For the SBC–11/21 parallel port, the level is fixed at 5.

Figure 6–8: DRV11–B Driver Prefix File (YBPFX.MAC) Excerpt

```
.title  YBPFX  - DRV11--B Device driver prefix module

;
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1985, 1986 by Digital Equipment Corporation.
; All rights reserved.
;

        .mcall  drvcf$
        .mcall  ctrcf$

        .macro  cross MAN, CYC
        .br;
        .br;
        .error ; IF YB$MAN = 1, YB$CYC MUST be = 1.
        .endc
        .endc
        .endm

              ; for cross-correlation of YB$MAN and YB$CYC

YB$PPR  ==      175.            ; Process priority
YB$IPR  ==      250.            ; Process initialization priority
YB$HPR  ==      4.              ; DRV11B hardware priority (do NOT change)

YB$FNC  ==      0.              ; value to set FNCT 1, FNCT 2 and FNCT 3 bits
         ; (allowable values for YB$FNC are all EVEN values 0.-14. inclusive)
YB$GO   ==      1.              ; value to set GO bit ( 0 or 1 ONLY)
YB$MAN  ==      1.              ; set to 1. if in maintenance (loop-back) mode.
                                ; set to 0. otherwise.
YB$CYC  ==      1.              ; Used to set the CYCL bit in the CSR. Set this
                                ; to 1. if the DMA transfers are to be initiated
                                ; by the Q-bus arbiter and set to 0. if the
                                ; DMA transfers are to be initiated by the user
                                ; device. MUST be set to 1. if YB$MAN is 1..

        drvcf$  dname=YB,nctrl=1
        ctrcf$  cname=A,nunits=1.,csrvec=<172414,124>,units=<0>
;       ctrcf$  cname=B,nunits=1.,csrvec=<172424,130>,units=<0>

{End of user-settable parameters}

        .end
```

YF$AIO defines the data direction for the 8-bit data port A; 0 = output, 1 = input. If the default direction of that port is reversed, jumpers M59 through M66 on the FALCON or FALCON–PLUS board must be changed from their factory configuration to reflect the prefix module settings.

YF$BIO defines the data direction for the 8-bit data port B; 0 = output, 1 = input. If the default direction of that port is reversed, jumpers M59 through M66 on the FALCON or FALCON–PLUS board must be changed from their factory configuration to reflect the prefix module settings.

Figure 6–9: SBC–11/21 PIO Driver Prefix File (YFPFX.MAC)

```
.title  YFPFX  - FALCON (SBC--11/21) 8255 PIO Device driver prefix module

;
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (C) 1982, 1986 by Digital Equipment Corporation.
; All rights reserved.
;

        .GLOBL  $YF             ; Haul in the driver from the library

YF$PPR  ==      175.            ; Process software priority
YF$IPR  ==      250.            ; Process initialization priority
YF$HPR  ==        5             ; Hardware interrupt priority

YF$AIO  ==        1             ; Port A is input
                                ; Set to 0 for Port A output
YF$BIO  ==        0             ; Port B is output
                                ; Set to 1 for Port B input

        .end
```

6.8.5 YK Prefix File

Figure 6–10 displays the YK prefix file driver YKPFX.MAC.

The following options are available when you configure the KXT11–CA or KXJ11–CA 8-bit parallel ports A and B:

- Port can be input, output, or bit mode. In bit mode, the direction of each bit is programmed individually.
- DMA transfers in fixed-length or stop-on-pattern mode. In stop-on-pattern mode, you can test data for specified patterns and can generate interrupt requests based on the match obtained.
- Programmable polarity—inverting or noninverting.
- Pulse catchers can be inserted into an input data path. When a pulse is detected at the pulse catcher input, its output is automatically set until it is cleared by the software's writing a nonpulse level to the corresponding bit in the data register. In all other cases, attempted writes to input bits are ignored. The pulse catcher is level-sensitive; therefore, if the impulse is still at the pulse level when it is cleared, the output will again become enabled.
- Optional open-drain outputs, with no pull-up resistors provided.
- Four handshake modes, including interlocked, strobed, pulsed, and 3-wire. When specified as a port with handshake, the transfer of data into or out of the port and interrupt generation is under handshake logic control.

  With interlocked handshake, the driver action must be acknowledged by the external device before the next action can take place. An output port does not indicate that new data is available until the external device indicates that it is ready for the data. Similarly, an input port does not indicate that it is ready for new data until the data source indicates that the previous byte of data is no longer available, thereby acknowledging the input port's

acceptance of the last byte. The handshake allows the YK driver to interface directly to a port, with no external logic.

With strobed handshake, the data is strobed by external logic into and out of the port. Unlike the interlocked handshake, the signal indicating that the port is ready for another data transfer operates independently of any input acknowledgment. The external logic must ensure that data does not transfer at a rate either too fast or too slow.

With a pulsed handshake, data is held for long periods of time and is gated with relatively wide pulses into or out of the driver. A pulsed handshake is used to interface to mechanical-type devices.

The 3-wire handshake is used when one output port is communicating with many input ports simultaneously. It is essentially the same as the interlocked handshake, except that two signals are used to indicate if an input port is ready for new data or if it has accepted the present data. With the 3-wire handshake, output lines on many input ports can be bused together with open-drain drivers; the output port knows when all ports have accepted the data and are ready. Because this handshake requires three lines, only one port—either A or B—can be a 3-wire handshake port at one time.

- Pattern- or transition-recognition logic. In ports A and B, you can test data for specified patterns and can generate interrupt requests based on the match obtained. The pattern-recognition logic is independent of the port application. The pattern can be independently specified for each bit as 1, 0, 0-to-1 transition, 1-to-0 transition, or any transition. Two modes of pattern-recognition operation are supported: AND and OR. Transition recognition is illegal on ports with handshake.
- Link option that provides one 16-bit port instead of two 8-bit ports.

In addition to the 8-bit parallel ports, the YK driver interfaces with a 4-bit special-purpose I/O port, which is available as a 4-bit parallel port if no handshake mode was selected for the 8-bit ports. Otherwise, this port provides the handshake signals.

The driver also interfaces with three independent 16-bit counter/timers with programmable output duty cycles—pulsed, 1-shot, and square wave—and up to four external access lines for each counter—count input, output, gate, and trigger. The counter/timer can be programmed to be retriggerable or nonretriggerable.

Many operations are possible, depending on how the timer is programmed. If the counter/timer's duty cycle is programmed in the pulse mode, the external "data available" output is initiated by the internal "data available" signal's being detected "TC" cycles before.

Note

"TC" is the value programmed in the counter/timer Time Constant register. The type of duty cycle—pulsed, 1-shot, or square-wave—determines how the pulsed handshake operates with a counter/timer that is being used as the "data available" output for the handshake.

If the counter/timer is programmed with the 1-shot duty cycle, the external "data available" output follows the internal "data available" signal as soon as it is detected.

If the counter/timer is programmed with a square-wave duty cycle, the external "data available" output follows the internal "data available" signal at a predetermined time (TC clock cycles after it is detected).

Note

The counter, gate, or trigger mode can be used only if the count bit used is available. The count bit must be specified to be an input, even if the port bit is programmed as an output bit, to allow the CPU to write that input directly.

In counter mode, the I/O line of the port associated with the counter/timer is used as an external counter input; in gate mode, it is used as an external gate input to the counter/timer; in trigger mode, it is used as a trigger input to the counter/timer.

When set to retrigger, each trigger causes the time constant value to be reloaded and a new countdown sequence to be initiated. When a counter/timer is programmed in square-wave mode, a retrigger will cause the time constant value to be reloaded and the new countdown to start on the first half of the square wave.

The KXT11-CA/KXJ11-CA parallel port and timer/counter prefix file uses four configuration macros. An initialization macro, YKCI$, defines symbols and other macros. The second, YKCP$, configures a port. The third, YKCT$, configures a timer. The fourth, YKCE$, marks the end of the configuration list. The second and third macros update internal symbols as they are used. The symbol values indicate what features have been selected. Those values are used to prevent the user from attempting to select an invalid configuration. Error messages are output to the listing file or terminal if a conflicting option is requested.

The configuration macros generate a data table that is used by the driver at system initialization time to configure the peripheral processor hardware registers.

Configuration Initialization Macro-YKCI$

Syntax

```
YKCI$
```

The configuration initialization macro YKCI$ must be the first macro used in the YK driver prefix file. YKCI$ has no parameters associated with it; it initializes symbols and defines the remainder of the macros and submacros to be used.

Port Configuration Macro-YKCP$

Syntax

```
YKCP$  chan=[A],ptype=[YK$BIT],hsh=[YK$INL],dskw=[0],out=[0],
       inv=[0],in1=[0],oco=[FALSE],plnk=[FALSE],dma=[FALSE],
       pat=[FALSE]
```

This macro is used to configure a particular port. The macro can be used by listing parameters in the order that they are defined or by using the KEYWORD=VALUE format, where the value can be a sum of bit definitions in the case of the direction, polarity, or one's catcher specifications. If all parameters for a port cannot fit on a single line, the macro can be reused for the same channel and the remainder of the parameters specified on the second usage. Any omitted parameter will take on the default value as specified above in square brackets. Each parameter is described below.

chan

Specifies the channel number to use. Permissible values are A, B, or C.

ptype

Specifies the port type: bit, input, or output. Permissible values are:

YK$BIT—bit port
YK$INP—input port
YK$OUT—output port

hsh

Specifies the type of handshake mode: interlocked, strobed, 3-wire, or pulsed.

YK$INL—interlock
YK$STR—strobed
YK$3WI—3-wire (IEEE 488)
YK$PUL—pulsed

Note

Timer 3 must be configured and a run-time request sent in order to use pulsed handshake. The timer set command must be sent prior to sending the first command to the port.

dskw

Specifies the deskew time, in cycles. Permissible values are 0, 2, 4, 6, 8, 10, 12, 14, or 16.

out

Specifies the I/O direction for each bit. If set, the bit is output. The following symbols may be OR'd together to define output bits.

Mnemonic	Value (octal)
YK$B0	1
YK$B1	2
YK$B2	4
YK$B3	10
YK$B4	20
YK$B5	40
YK$B6	100
YK$B7	200

inv

Specifies polarity of the bit. If a bit is set, the corresponding input or output bit of the interface register is inverted. The symbols YK$B0 through YK$B7 listed above may be OR'd together to define inverted bits.

in1

Specifies which input bits should have the one's catcher attribute. If a bit is set, the corresponding input bit of the interface register will have the one's catcher enabled. The symbols YK$B0 through YK$B7 listed above may be OR'd together to define one's catcher bits.

oco

Specifies which output bits have open drain. If not specified, outputs will be active pull up.

plnk

If specified as TRUE, ports A and B will be linked together to form one 16-bit port.

dma

If specified, the port will use DMA.

pat

If specified, the port will use pattern recognition.

Timer Configuration Macro-YKCT$

This macro is used to configure a particular timer. The macro can be used by listing the parameters in the order that they are defined or by using the KEYWORD=VALUE format. If all parameters to be specified for a timer cannot fit on a single line, the macro can be reused for the same timer number and the remainder of the parameters specified on the second usage. Any parameter omitted takes on the default value specified as follows:

Syntax

```
YKCT$  tnum=[1],texto=[NO],textc=[NO],textt=[NO],textg=[NO],
       tretre=[NO],tout=[YK$TPL],tlnk=[YK$TIN]

tnum
       Defines the timer number: 1, 2, or 3

texto

       NO  --  Disables timer output
       YES --  Enables timer output

textc

       NO  --  Disables timer external count
       YES --  Enables timer external count

textt

       NO  --  Disables timer external trigger
       YES --  Enables timer external trigger

textg

       NO  --  Disables timer external gate
       YES --  Enables timer external gate

tretre

       NO  --  Disables timer retrigger
       YES --  Enables timer retrigger

tout
       Defines the type of timer output
```

```
        YK$TPL --  Pulse output
        YK$T1S --  One shot
        YK$TSQ --  Square wave

tlnk
        Defines the interaction of timers 1 and 2

        YK$TIN --  Timers are independent
        YK$1G2 --  Timer 1 output gates timer 2
        YK$1T2 --  Timer 1 output triggers timer 2
        YK$1I2 --  Timer 1 output is timer 2 input
```

End Configuration Macro-YKCE$

This macro must be used after all port or timer macros have been used. The end macro builds the configuration table from the local symbols that were defined while the other two macros were being used.

Parameter Interaction

Many inputs, outputs, and internals in the parallel port and timer/counter chip are multiplexed among the various functions. Thus, several features are mutually exclusive or are not available for particular ports or combinations of ports.

Error messages can occur during assembly of the YK prefix file if a chosen combination of parameters is not a viable configuration. See the appropriate *MicroPower/Pascal Messages Manual* for a list of the possible error messages.

The tables below show which combinations of parameters are invalid for the YKCP$ macro when configuring ports A, B, and C. If marked with an X, the parameter combination is invalid for the port. For example, when configuring port A, you cannot specify an in1 value if you specify ptype=YK$INP, YK$OUT, or YK$BID. The tables do not consider invalid combinations among ports; see the configuration notes for those.

General Port and Timer Configuration Notes

Handshake Signals

- If timer 3 has external output, bit 0 of port C cannot be a handshake signal.
- Only a single port can specify pulsed handshake.
- Only a single port can specify 3-wire handshake.
- If one port uses 3-wire handshake, other port must be a bit port.

Port Outputs

- Output lines of ports A and B must all be open collector or all be active pull-up.
- If timer 1 has external output, bit 4 of port B must be an output.
- If timer 2 has external output, bit 0 of port B must be an output.
- If timer 3 has external output, bit 0 of port C must be an output.

Timer External Outputs

- Port B must be a bit port to use timer 1 or timer 2 external output.
- If timer 1 has external output, bit 4 of port B must be an output.

- If timer 2 has external output, bit 0 of port B must be an output.
- Port C must be configured in order to use timer 3 external output.
- To use external output for timer 3, port C must be configured.
- If timer 3 has external output, bit 0 of port C must be an output.
- If timer 3 has external output, bit 0 of port C cannot be a handshake signal.

Invalid YKCP$ Parameter Combinations for Port A

port A	YK$BIT	YK$INP	YK$OUT	YK$INL	YK$STR	YK$PUL	YK$3WI	dskw	out	inv	in1	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X					X		
YK$INP	X		X					X	X		X				
YK$OUT	X	X							X		X				
YK$INL	X				X	X	X								
YK$STR	X			X		X	X								
YK$PUL	X			X	X		X								
YK$3WI	X			X	X	X									
dskw	X	X													
out		X	X												
inv															
in1		X	X												
oco															
plnk	X														
dma															
pat															

MLO-880-87

Port A Configuration Notes

- If port A is a bit port (ptype=YK$BIT), bits 0–3 and bits 4–7 must all be inputs or all be outputs. Thus, if port A is a bit port, the parameter **out** can have only the values 0, 17, 360, or 377 (octal).
- If ports A and B are linked, port A cannot be a bit port.
- To use handshake signals on port A, port C must be configured.
- The value of oco for ports A and B must be the same.

Invalid YKCP$ Parameter Combinations for Port B

port B	YK$ BIT	YK$ INP	YK$ OUT	YK$ INL	YK$ STR	YK$ PUL	YK$ 3WI	dskw	out	inv	in1	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X							
YK$INP	X		X					X	X		X		X		
YK$OUT	X	X							X		X		X		
YK$INL	X				X	X	X								
YK$STR	X			X		X	X								
YK$PUL	X			X	X		X								
YK$3WI	X			X	X	X									
dskw	X	X													
out		X	X												
inv															
in1		X	X												
oco															
plnk		X	X												X
dma															
pat													X		

MLO-881-87

Port B Configuration Notes

- If ports A and B are linked, port B must be a bit port.
- To use handshake signals on port B, port C must be configured.
- The value of oco for ports A and B must be the same. That is, the output lines of ports A and B must all be open collector or all be active pull-up.

Invalid YKCP$ Parameter Combinations for Port C

port C	YK$BIT	YK$INP	YK$OUT	YK$INL	YK$STR	YK$PUL	YK$3WI	dskw	out	inv	in1	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X					X	X	X
YK$INP	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
YK$OUT	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
YK$INL	X	X	X		X	X	X	X					X	X	X
YK$STR	X	X	X	X		X	X	X					X	X	X
YK$PUL	X	X	X	X	X		X	X					X	X	X
YK$3WI	X	X	X	X	X	X		X					X	X	X
dskw	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
out		X	X					X					X	X	X
inv		X	X					X					X	X	X
in1		X	X					X					X	X	X
oco		X	X					X					X	X	X
plnk	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
dma	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
pat	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

MLO-882-87

Port C Configuration Notes

- Port C is four bits wide; therefore, values for inv, in1, and out can have values only in the range 0 to 17 (octal).
- To use handshake signals on ports A and B, port C must be configured.
- Port C handshake inputs cannot be defined as outputs.
- Port C handshake outputs cannot be defined as inputs.

Figure 6–10: KXT11–CA/KXJ11–CA PIO Driver Prefix File (YKPFX.MAC)

```
        .TITLE  YKPFX - Parallel I/O and counter/timer Driver Prefix Module
        .ident  /V2.0/
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.
;
;+
;  This module is an example of using the special configuration
; macros for the Parallel I/O and Counter-Timers on the KXT 11C.
; The header at the beginning of the module lists the features that are
; being configured.
;
;  This configuration provides:
;
;          1.  4 switch inputs
;          2.  4 LED driver outputs
;          3.  8 line parallel output port, with pattern match or DMA
;          4.  2 pulsed handshakes (1 input, 1 output) for output port
;          5.  1 status input line from parallel device
;          6.  Timer 3 provides delay time in pulsed handshake
;          7.  Timer 1 and 2 are linked (timer 1 = least sig word)
;-

;+
;DEFINE PRIORITIES FOR YK HANDLER
;-
YK$HPR  == 5            ; Hardware priority
YK$IPR  == 250.         ; Initialization priority
YK$PPR  == 180.         ; Process priority

;+
;CALL:  INITIALIZE MACRO
;-
        .MCALL  YKCI$
        YKCI$
;+
; PORT A
; 'bit port' for reading switches and driving LEDs
; bits 0,1,2,3 are inputs.... bits 4,5,6,7 are inverted outputs
;-
        YKCP$   CHAN=A,PTYPE=YK$BIT,OUT=<YK$B4+YK$B5+YK$B6+YK$B7>
        YKCP$   CHAN=A,INV=<YK$B4+YK$B5+YK$B6+YK$B7>

;+
; PORT B
; parallel output port, with pulsed handshake
;-
        YKCP$   CHAN=B,PTYPE=YK$OUT,HSH=YK$PUL,PAT=YES,DMA=YES
```

```
;+
; PORT C
;  handshake signals for port B
;  bit 0 = acknowledge (input)
;  bit 1 = data available (inverted open collector output)
;
;  status input from external device
;  bit 2 = non-inverted input
;-
        YKCP$  CHAN=C,OUT=<YK$B1>,INV=YK$B1,OCO=YES

;+
; TIMER 1
;-
        YKCT$  TNUM=1

;+
; TIMER 2
; timer 1 output is timer 2 input
;-
        YKCT$  TNUM=2,TLNK=YK$1I2

;+
; TIMER 3
;-
        YKCT$  TNUM=3

;+
; END CONFIGURATION
;-
        YKCE$

        .END
```

Chapter 7
Analog-to-Digital Converter Driver

This chapter describes the use of the MicroPower/Pascal analog-to-digital converter (AD) driver, which supports I/O operations on the ADV11–C and AXV11–C analog-to-digital (A/D) converter boards. The ADV11–C and AXV11–C devices interface analog inputs to a MicroPower/Pascal target processor, so that A/D conversions can be performed.

The chapter also describes a Pascal routine, WRITE_ANALOG_WAIT, that supports programmed I/O on the AAV11–C and AXV11–C digital-to-analog (D/A) converters. The devices AAV11–C and AXV11–C D/A interface analog outputs to a target, so that D/A conversions can be performed.

7.1 Driver Features and Capabilities

The AD driver supports A/D conversion setup, reading of converted values, and returning of device characteristics.

The A/D conversion setup operation sets up and enables A/D conversion, specifying the method of triggering the conversion, the number of analog input channels to sample and their identifying numbers, and the gain on each channel. Optionally, you can set a limit on the total number of records (sets of samples) to return, after which read requests will be ignored. A/D conversions can be started by a read request (immediate triggering), an external event trigger, or a real-time clock input.

After the setup operation, the driver and device are ready for conversions to be triggered in the specified manner. If external event or real-time clock triggering is in effect, A/D conversions occur asynchronously with respect to user requests for converted data. Converted data is transferred from the device to the driver's buffers by the driver's interrupt service routine (ISR). Once in the ISR buffers, the data awaits user requests for transfer to user buffers.

A read operation returns one or more records to a user-specified buffer. If external or real-time clock triggering is in effect, the read returns records from the asynchronous device-to-ISR input stream; if none are available, the driver waits until externally-triggered conversions generate more records. The size of the user buffer—a multiple of the number of channels sampled, times two (for byte units)—determines the number of returned records. If immediate triggering is in effect, the read request triggers the conversion and returns a single record to the user buffer.

Get Characteristics returns codes for device class and type.

7.2 Performing Analog-to-Digital Conversions

For most MicroPower/Pascal applications, you perform A/D conversions in one of two ways:

1. You can invoke Pascal I/O procedures that open files for converted data and then input the data in accordance with the rules for standard Pascal I/O. The Pascal I/O procedures—OPEN, GET, READ, and so forth—are described in Chapter 9 of the *MicroPower/Pascal Language Guide*. Note that this method requires that you call the support routine SET_ANALOG_MODE to supply the necessary conversion control information before reading data. (See item 2.)
2. You can invoke the Pascal support routines SET_ANALOG_MODE and READ_ANALOG_SIGNAL. Those routines provide high-level nonfile access to the A/D converters. The A/D support routines issue Pascal SEND requests to the request queue semaphore of the AD driver. The routines are described in Section 7.4.

Note

You can perform D/A conversions by using the WRITE_ANALOG_WAIT support routine. (See Section 7.4.)

In addition to invoking the Pascal I/O procedures or A/D support routines, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the A/D controller interrupt vector addresses
2. Edit the AD driver prefix file to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, number of controller units (1) and identifying number (0), and the ISR buffer size
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - AD driver process
   - [For real-time clock triggering:] KW driver process (see Chapter 8)
   - [For A/D conversion:] A/D support routine SET_ANALOG_MODE (from kit files ADINC.PAS and ADSUB.PAS)
   - [For nonfile access:] A/D support routine READ_ANALOG_SIGNAL and/or D/A programmed I/O routine (from kit files ADINC.PAS and ADSUB.PAS)
   - [For file OPEN:] Ancillary control process (ACP)
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any I/O support routines you choose to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for A/D conversions, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 7.7 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

Alternatives to using the A/D support routines or the Pascal I/O procedures for A/D conversions exist, but require more effort. You can:

- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and the driver, bypassing the OTS file routines (lower-level file system access).
- Issue your own Pascal or MACRO–11 packet-level requests to the driver, bypassing the OTS file routines, the ACP, and the A/D support routines (low-level nonfile access).

The following sections describe the Pascal I/O procedure interface to the AD driver, the Pascal support routine interface, the lower-level request/reply packet interface, the status codes that can be returned to users of any interface, and the AD driver prefix file.

7.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O for A/D conversion data, you must open a file. Opening the file associates a Pascal file variable with an A/D converter board. Invoke the OPEN procedure as follows:

OPEN (filvar, 'ADc0:', ...)

where:

- filvar is a Pascal file variable.
- c is a controller identifier (A, B, ...).

For example, 'ADA0:' would specify unit 0 of the first A/D converter (A) listed in the AD driver prefix file.

The OPEN causes the Pascal OTS to send a packet-level open request to the ACP, which returns a unit number and a driver request semaphore ID to the OTS. Subsequent I/O requests are sent directly to the driver by the OTS, with no further ACP involvement.

After the OPEN and before reading data, you must call the support routine SET_ANALOG_MODE in order to set up and enable A/D conversions. SET_ANALOG_MODE is described in Section 7.4.1.

In carrying out subsequent input, CLOSE, or PURGE operations on A/D converters, the Pascal OTS uses the following packet-level driver functions:

- Read Logical (IF$RDL)
- Close (IF$CLS)
- Purge (IF$PRG)

The appropriate request packets are sent to the driver only when necessary to complete a user-requested operation. For example, a READ or GET operation that requires more data than what remains in the buffers from previous input operations causes the OTS to issue one or more Read Logical (IF$RDL) requests to the AD driver.

Pascal Get Characteristics functions are provided in the file GETSET.PAS on the MicroPower/Pascal distribution kit. Those functions issue Get Characteristics (IF$GET) request packets to the driver.

7.4 Pascal Support Routine Interface

The following support routines, written in Pascal and independent of the file system, are provided as an alternative high-level interface to the A/D and D/A hardware:

- SET_ANALOG_MODE
- READ_ANALOG_SIGNAL
- WRITE_ANALOG_WAIT

Note

The A/D routines SET_ANALOG_MODE and READ_ANALOG_SIGNAL use all of the packet-level AD driver functions except Get Characteristics (IF$GET). To perform that operation, use the Get Characteristics function (descriptor version) in the kit file GETSET.PAS.

The SET_ANALOG_MODE routine, although not file-oriented, is required for the Pascal I/O procedure (file system) interface as well as the Pascal support routine interface.

The following sections describe the Pascal routines for A/D and D/A I/O. The A/D routines SET_ANALOG_MODE and READ_ANALOG_SIGNAL each allocate an I/O packet, fill it in with information based on the function parameters, send it to the AD driver queue semaphore, and return immediately to the caller. If the routine has a reply parameter, the driver sends a standard driver reply to the specified queue semaphore when the operation is complete. (The driver reply packets are described in Section 7.5.)

The D/A routine WRITE_ANALOG_WAIT uses programmed I/O rather than the AD driver. It writes values from a buffer to one or more D/A channels.

The following files on the MicroPower/Pascal distribution kit are required for using the routines:

File	Description
ADSUB.PAS	Analog I/O routine source module
ADINC.PAS	Analog I/O routine include file
IOPKTS.PAS	Pascal I/O include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

7.4.1 SET_ANALOG_MODE

The SET_ANALOG_MODE procedure sets up and enables A/D conversion, specifying the method of triggering the conversion, the number of analog input channels to sample and their identifying numbers, and the gain on each channel. Optionally, you can set a limit on the total number of records (sets of samples) to return, after which read requests will be ignored.

A/D conversions can be started by a read request (immediate triggering), an external event trigger, or a real-time clock input. The read request or external input initiates the analog conversion of the first channel of each sample, following which the A/D converter requests an interrupt. The driver's ISR then initiates conversion of the second to nth channels.

External event triggering assumes either that an external event signal, asserted low, is connected to EXT IN L at the I/O connector on the board and jumper F2 is connected or that jumper F1 is connected for the LSI-11 bus (Q-bus) BEVNT line.

Real-time clock triggering assumes that the real-time clock (KWV11-C) is present and that the clock output signal (CLK OVL—clock overflow, asserted low) is connected to RTC IN L (real-time clock input, asserted low) on the A/D board.

After the SET_ANALOG_MODE operation, the driver and device are ready for conversions to be triggered in the specified manner. If external event or real-time clock triggering is in effect, A/D conversions occur asynchronously with respect to user requests for converted data. Converted data is transferred from the device to the driver's buffers by the ISR. Once in the ISR buffers, the data awaits user requests for transfer to user buffers.

The packet-level equivalent of SET_ANALOG_MODE is the IF$SET function.

The syntax for calling the procedure is as follows:

```
SET_ANALOG_MODE ( buffer, trigger, count, ad_desc, reply );
```

Parameter	Type	Description
VAR buffer	AD_CONTROL_TYPE	User-constructed channel parameter block specifying the number of channels to be sampled and the identifying number and gain for each channel.
trigger	INTEGER	Value specifying the method for triggering A/D conversions—CONVRT_IMED (0) for immediate triggering on receipt of a read request, EXTERNAL_EVENT (16.) for external event triggering, or REAL_TIME_CLOCK (32.) for real-time clock triggering. (The values are defined in ADINC.PAS and correspond to CSR bit positions.)
count	INTEGER	Optional specification of the total number of records to be returned, after which read requests are ignored; the default is 0, which sets up a continuous read operation.

Parameter	Type	Description
VAR ad_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor.
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply.

The data type AD_CONTROL_TYPE, from ADINC.PAS, is shown below:

```
TYPE
  ad_gain = (
    ad_gain_1,
    ad_gain_2,
    ad_gain_4,
    ad_gain_8 );

  mpx_addr = 0..15;   { Channel number }

   ad_chan_desc = PACKED RECORD
     chan_num    :   [BIT(12)] mpx_addr;
     gain_sel    :   [BIT(4)] ad_gain;
   END;

   ad_control_type = RECORD
     num_chan    :   INTEGER;
     chan_ctrl    :   ARRAY [1..16] OF ad_chan_desc;
   END;
```

NUM_CHAN specifies the number of channels to be sampled.

CHAN_NUM designates the analog input channel (multiplexer address) to be sampled. The channel number selects either one of 16 single-ended analog input channels or one of eight differential input channels. Whether the analog input is single-ended or differential is determined by the installed type of jumper (SI/DI).

GAIN_SEL specifies a gain-select value. The gain corresponding to the possible gain-select values are shown below:

Value	Gain	Range
AD_GAIN_1	1	10 V
AD_GAIN_2	2	5 V
AD_GAIN_4	4	2.5 V
AD_GAIN_8	8	1.25 V

Indication of success or failure of the setup operation is returned in the status field of the AD driver reply packet.

Note

The AD_CONTROL_TYPE data structure and its conversion control information are diagrammed in Section 7.5.1.

7.4.2 READ_ANALOG_SIGNAL

The READ_ANALOG_SIGNAL procedure returns records—sets of converted data—to a user-specified buffer.

If external or real-time clock triggering is in effect, READ_ANALOG_SIGNAL returns records from the asynchronous device-to-ISR input stream; if none are available, the driver waits until externally-triggered conversions generate more records. The size of the user buffer—a multiple of the number of channels sampled, times two (for byte units)—determines the number of records returned.

If immediate triggering is in effect, READ_ANALOG_SIGNAL triggers the conversion and returns a single record to the user buffer.

The packet-level equivalent of READ_ANALOG_SIGNAL is the IF$RDL function.

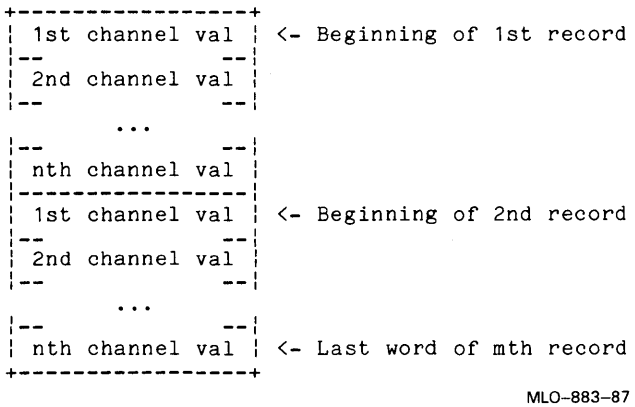
The syntax for calling the procedure is as follows:

```
READ_ANALOG_SIGNAL ( buffer, ad_desc, reply );
```

Parameter	Type	Description
VAR buffer	ARRAY[first..last: INTEGER] OF INTEGER	Buffer to which one or more records will be returned
VAR ad_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

Each converted data value is the result of the A/D conversion on the specified channel and gain. The range of the converted data values depends on the jumper configuration on the boards. The range of values may be from –4000 (octal) to 3777 (octal) or from 0 to 7777 (octal). See the hardware user's guide for details on setting the jumpers.

The converted data is stored in the specified buffer, which must begin on a word boundary, as follows:



MLO-883-87

The count of transferred bytes is returned in the actual-length field of the AD driver reply packet.

7.4.3 WRITE_ANALOG_WAIT

The WRITE_ANALOG_WAIT procedure supports D/A conversion via programmed I/O transfer. The procedure does not use a device driver. WRITE_ANALOG_WAIT interfaces with the AAV11-C and AXV11-C D/A converters. It writes one to four (AAV11-C) or one to two (AXV11-C) values from a buffer to one or more D/A channels. WRITE_ANALOG_WAIT requires that the calling process have I/O page access.

The syntax for calling the procedure is as follows:

```
WRITE_ANALOG_WAIT ( channels, buffer, state );
```

Parameter	Type	Description
VAR channels	ARRAY[chan1.. num_chan:dac_chan] OF INTEGER	Array specifying channels for the corresponding entries in the "buffer" array to be written to; buffer[n] is written to channels[n]
VAR buffer	ARRAY[val1.. num_values: dac_chan] OF INTEGER;	Array of integers to be written to D/A converters specified in the "channels" array
VAR state	UNSIGNED	Location to which to return status code; a returned value of 1 signals success, and a returned value of -1 indicates an invalid parameter

The data type DAC_CHAN, from ADINC.PAS, is shown below:

```
TYPE
  dac_chan = 0..3;
```

7.5 Request/Reply Packet Interface

The packet-level functions provided by the AD driver are listed below by symbolic and decimal function code:

Code	Function
IF$RDL (1)	Read Logical (Read Converted Data)
IF$SET (6)	Set Characteristics (Configure Device)
IF$GET (7)	Get Characteristics

If a request is received for an Open (IF$LOK or IF$ENT), a Close (IF$CLS), or a Purge (IF$PRG), the driver returns an illegal function status code (ES$IFN), which the ACP (Open) or OTS (Close/Purge) interprets as indicating that no device-dependent processing was required for that operation.

Note

The MACRO–11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

A single function modifier is recognized by the AD driver, as shown below:

Code	Function
FM$BSM (bit 13)	Signal binary/counting semaphore

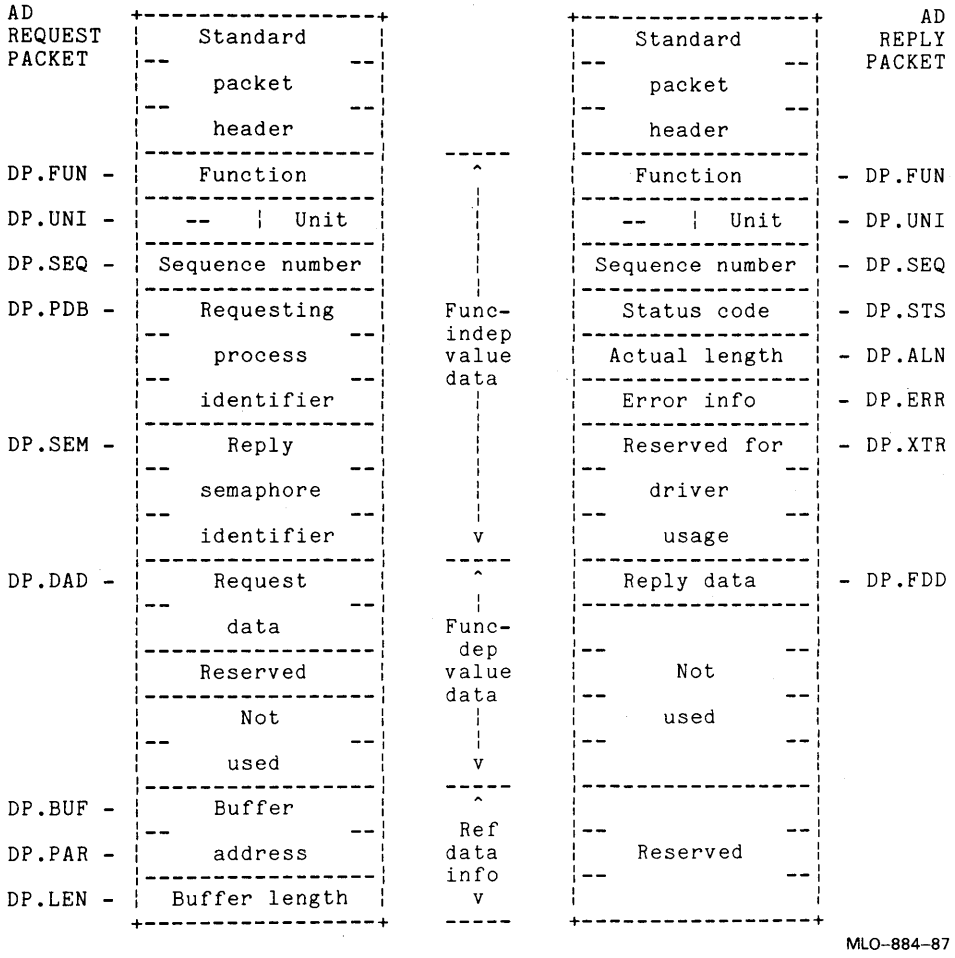
The AD driver consists of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each controller configured. I/O requests for a controller are sent (using a Pascal SEND or a MACRO–11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The request queue names and number of supported units for AD driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
A/D converter	$ADc	1	0

The letter c in a queue name represents a controller designation (A, B, ..., as specified in an AD driver prefix file).

The general format of the A/D request and reply packets is shown below:



The function-independent portions of the packets shown above are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit number for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of AD driver function.

Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

7.5.1 Set Characteristics (Configure Device) Function

The Set Characteristics (IF$SET) function sets up and enables A/D conversion, specifying the method of triggering the conversion, the number of analog input channels to sample and their identifying numbers, and the gain on each channel. Optionally you can set a limit on the total number of records (sets of samples) to return, after which read requests will be ignored.

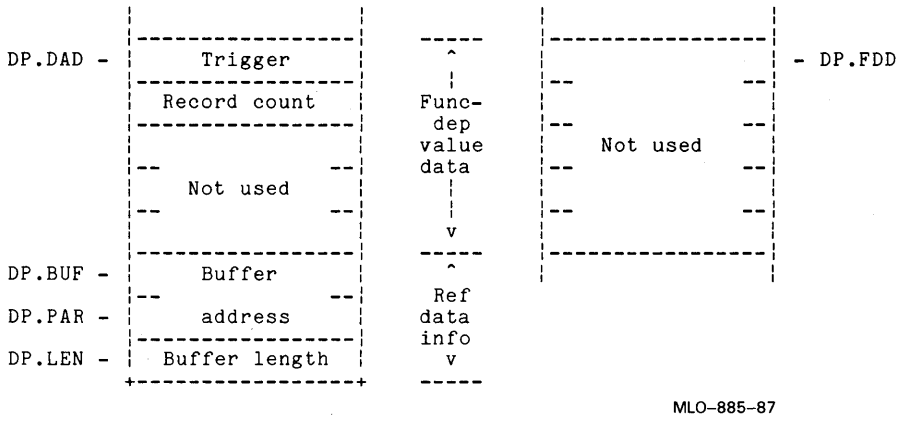
A/D conversions can be started by a read request (immediate triggering), an external event trigger, or a real-time clock input. The read request or external input initiates the analog conversion of the first channel of each sample, following which the A/D converter requests an interrupt. The driver's ISR then initiates conversion of the second to nth channels.

External event triggering assumes either that an external event signal, asserted low, is connected to EXT IN L at the I/O connector on the board and jumper F2 is connected or that jumper F1 is connected for the LSI-11 bus (Q-bus) BEVNT line.

Real-time clock triggering assumes that the real-time clock (KWV11-C) is present and that the clock output signal (CLK OVL—clock overflow, asserted low) is connected to RTC IN L (real-time clock input, asserted low) on the A/D board.

After the Set Characteristics operation, the driver and device are ready for conversions to be triggered in the specified manner. If external event or real-time clock triggering is in effect, A/D conversions occur asynchronously with respect to user requests for converted data. Converted data is transferred from the device to the driver's buffers by the ISR. Once in the ISR buffers, the data awaits user requests for transfer to user buffers.

The function-dependent portions of the Set Characteristics request and reply packets are shown below:

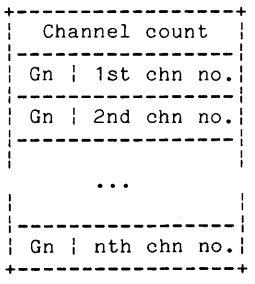


MLO-885-87

The trigger word specifies the method for initiating an A/D conversion. The value 0 specifies immediate conversion on receipt of a read request, 16 (bit 4 set) specifies external triggering, and 32 (bit 5 set) specifies real-time clock triggering. (The values correspond to CSR bit positions.)

The record count specifies the total number of records (sets of converted data) to be returned, after which end-of-file is considered to have been reached. Read requests received after that point are ignored. A record count of 0 sets up a continuous read operation.

The buffer-address and buffer-length fields specify the location and length of a user-constructed channel parameter block that gives control information for the conversion of the digitized data. The control information consists of a count of the number of channels to be sampled, plus a descriptor word for each channel, as shown below:



MLO-886-87

The 4-bit Gn field specifies a gain-select value, an unsigned binary integer in the range 0 through 3. The gain corresponding to the possible gain-select values are shown below:

Value	Gain	Range
0	1	10 V
1	2	5 V
2	4	2.5 V
3	8	1.25 V

The channel number is a 12-bit unsigned integer in the range 0 through 15 designating the analog input channel (multiplexer address) to be sampled. It selects either one of 16 single-ended analog input channels or one of eight differential input channels. Whether the analog input is single-ended or differential is determined by the installed type of jumper (SI/DI).

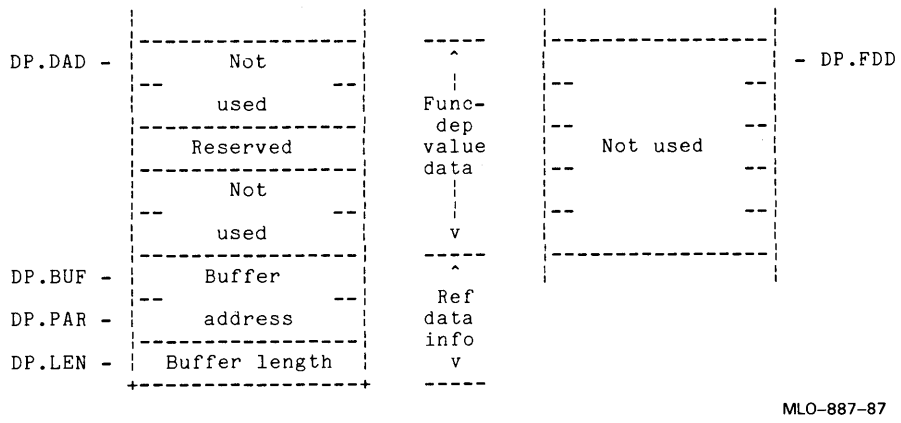
7.5.2 Read Logical (Read Converted Data) Function

The read (IF$RDL) function returns records—sets of converted data—to a user-specified buffer.

If external or real-time clock triggering is in effect, the read returns records from the asynchronous device-to-ISR input stream; if none are available, the driver waits until externally-triggered conversions generate more records. The size of the user buffer—a multiple of the number of channels sampled, times two (for byte units)—determines the number of records returned.

If immediate triggering is in effect, the read request triggers the conversion and returns a single record to the user buffer.

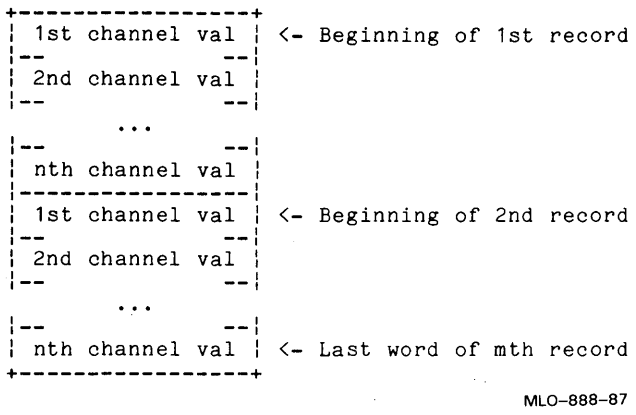
The function-dependent portions of the read request and reply packets are shown below:



MLO-887-87

Each converted data value is the result of the A/D conversion on the specified channel and gain. The range of the converted data values depends on the jumper configuration on the boards. The range of values may be from -4000 (octal) to 3777 (octal) or from 0 to 7777 (octal). See the hardware user's guide for details on setting the jumpers.

The converted data is stored in the specified buffer, which must begin on a word boundary, as follows:



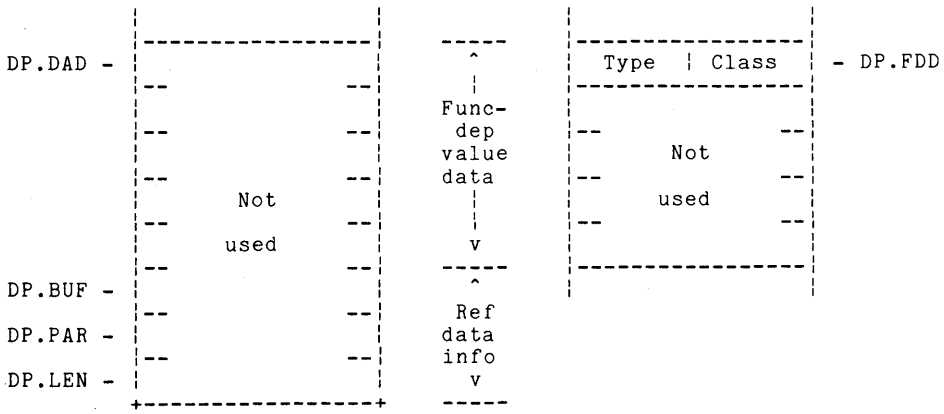
MLO-888-87

The count of transferred bytes is returned in the actual-length field of the AD driver reply packet.

7.5.3 Get Characteristics Function

The A/D Get Characteristics (IF$GET) function returns, in the function-dependent portion of the reply message, the codes for A/D device class and type.

The function-dependent portions of the A/D Get Characteristics request and reply packets are shown below:



MLO-889-87

In the preceding information:

- Class is DC$RLT for real-time device class.
- Type is RT$ADV for the ADV11-C, RT$AXV for the AXV11-C.

7.6 Status Codes

If an A/D device or the AD driver detects an error during an I/O operation, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error is detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply message.

The AD driver returns the following exception codes:

Code	Type	Description
ES$IVP	HARD_IO	Invalid parameter: negative channel count, channel or gain out of range
ES$IFN	SOFT_IO	Illegal function; also used internally to signal ACP or OTS that no device-dependent processing of an Open, Close, or Purge was required

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for OTS-detected I/O errors or for kernel-detected errors that the AD driver raises rather than passing back to the requesting process. OTS-detected I/O errors are listed in Chapter 9 of the *MicroPower/Pascal Language Guide*.

7.7 AD Driver Prefix File

Figure 7–1 shows the AD driver prefix module. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The symbols AD$IPR, AD$PPR, and AD$HPR define the initialization and request-handling software priorities for the driver process and the hardware interrupt priority for the controller(s).

The DRVCF$ macro contains a field for the number of controllers on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced by the driver. It gives the controller name, number of units (1), CSR and vector addresses, unit number (0), and ISR buffer size.

The ISR buffer size—normally a multiple of the record size (as determined by a SET_ANALOG_MODE or IF$SET operation)—applies to each of two internal buffers that receive converted data from the device for transfer to user buffers. The driver swaps the buffers as necessary to maintain a steady flow of data.

The interrupt vectors must also be specified in the system configuration file, using the DEVICES macro.

Figure 7–1: AD Driver Prefix File (ADPFX.MAC)

```
.title   ADPFX   - A/D Converter Prefix Module

;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1984, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS
; RESERVED.
;

   .mcall   drvcf$
   .mcall   ctrcf$
   .mcall   adisz$

   adisz$

AD$PPR   ==   184.       ; Process priority
AD$HPR   ==   4          ; A/D hardware priority
AD$IPR   ==   250.       ; Process initialization priority

   drvcf$   dname=AD,nctrl=1
   ctrcf$   cname=A,nunits=1.,csrvec=<170400,400>,units=<0:0>,typrm=64.
   .end
```

Chapter 8
Real-Time Clock Driver

This chapter describes the use of the MicroPower/Pascal real-time clock (KW) driver, which supports I/O operations on the KWV11–C programmable real-time clock.

The KWV11–C can be programmed to count from one of five crystal-controlled frequencies, from an external input frequency, from an external event or number of events, or from the 50/60 Hz line frequency on the LSI–11 bus (Q-bus). The clock can generate interrupts or can synchronize the processor to external events. The clock has a counter that can be programmed to operate in any of the following modes: single interval, repeated interval, external event timing, or external event timing from zero base.

The KWV11–C clock has two Schmitt triggers. In response to external events, they can start the clock, start analog-to-digital (A/D) conversions in an A/D converter (see Chapter 7), or generate program interrupts to the processor.

An A/D conversion may be started at a crystal-controlled rate, at a line frequency rate (50/60 Hz), or from an external event input.

8.1 KW Driver Features and Capabilities

The KW driver supports reading, enabling, and disabling of the KWV11–C real-time clock and returning of standard device characteristics.

Read operations support the external event modes of the KWV11–C. You can record the time of external events or the time between external events. In addition, two events can be monitored with respect to each other.

The clock enabling operation sets up the clock for interval timing or as a free-running clock used to initiate A/D conversions.

The clock disabling operation stops the clock by disabling interrupts on the device.

The Get Characteristics operation returns codes for device class and type.

8.2 Performing Real-Time Clock I/O

For most MicroPower/Pascal applications, you perform real-time clock I/O by invoking the Pascal support routines READ_COUNTS_WAIT, READ_COUNTS_SIGNAL, START_RTCLOCK, and STOP_RTCLOCK. Those routines provide high-level nonfile access to a clock. The KW support routines issue Pascal send requests to the request queue semaphore of the KW driver. The routines are described in Section 8.3 (Pascal Support Routine Interface).

Note

DIGITAL recommends that you do not perform file-oriented operations on a real-time clock. Although the KW driver does not prevent you from opening a file for clock data, the operation is of little use; the KW read, clock enabling and disabling, and Get Characteristics operations cannot be performed with standard Pascal I/O statements, such as GET and WRITE.

In addition to invoking the KW support routines, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the KW controller interrupt vector addresses
2. Edit the KW driver prefix file to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, number of controller units (1) and identifying number
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - KW driver process
   - [For A/D conversion triggering:] AD driver process (see Chapter 7)
   - Pascal real-time clock support routines (from kit files KWSUB.PAS and KWINC.PAS)

For more information on setting up your application software for real-time clock I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 8.6 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

As an alternative to using the Pascal support routines described in this chapter to perform real-time clock I/O, you can issue your own Pascal or MACRO–11 packet-level requests to the driver (low-level nonfile access).

The following sections describe the Pascal support routine interface to the KW driver, the lower-level request/reply packet interface, the status codes that can be returned to users of either interface, and the KW driver prefix file.

8.3 Pascal Support Routine Interface

The following support routines, written in Pascal and independent of the file system, provide a high-level interface to the real-time clock hardware:

- READ_COUNTS_WAIT
- READ_COUNTS_SIGNAL
- START_RTCLOCK
- STOP_RTCLOCK

Note

The real-time clock support routines perform all the packet-level KW driver functions except Get Characteristics (IF$GET). See the kit file GETSET.PAS for a nonfile-oriented Get Characteristics function.

The following sections describe the Pascal routines for real-time clock I/O. Each routine allocates an I/O packet, fills it in with information based on the procedure parameters, and sends it to the KW driver queue semaphore. Most of the routines then issue a RECEIVE request for the KW driver reply and return to the caller after the reply is received. However, the READ_COUNTS_SIGNAL routine returns to the caller immediately after SENDing to the driver. If a reply parameter was specified on the READ_COUNTS_SIGNAL call, the driver sends a standard driver reply via the specified queue semaphore when the operation is complete. (The driver reply packets are described in Section 8.4.)

The following files on the MicroPower/Pascal distribution kit are required for using the routines:

File	Description
KWSUB.PAS	Real-time clock routine source module
KWINC.PAS	Real-time clock routine include file
IOPKTS.PAS	Pascal I/O include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

8.3.1 READ_COUNTS_WAIT

The READ_COUNTS_WAIT procedure reads a block of counts from the programmable real-time clock. It issues a Read Physical request to the KW driver and waits until it is completed to return. READ_COUNTS_WAIT can be used to record the time of external events or the time between external events. Also, two events can be monitored with respect to each other.

External events are detected by Schmitt triggers, two of which reside on the KWV11-C board. The primary Schmitt trigger is the second one, ST2. It can be used to cause interrupts or to start the clock. The first Schmitt trigger, ST1, can be used only to increment the clock.

To record the time of external events or the time between external events, a fixed rate is specified for the source. The clock is started by either the driver setting the GO bit in the CSR or by the first external event on Schmitt trigger 2. When the clock starts, the clock counter is cleared and subsequently incremented at the specified rate. When an event occurs on Schmitt trigger 2, the

value of the counter is transferred to the buffer/preset register, and an interrupt is requested. If zero-base is specified, the counter is zeroed; otherwise, the clock is incremented from its current value. (Continuous incrementing gives the time of the event, and zero-base gives the time between events.) The ISR reads the count from the buffer/preset register and copies it to the user-specified buffer. An overrun condition occurs when a second external event occurs before the ISR has read the count from the preceding external event and indicates that the events are occurring too quickly for the system to handle. An overflow condition occurs when the clock counter overflows before a second external event occurs and may indicate that too high a rate was specified.

To measure the relative frequency of one event to another, proceed as above, but specify Schmitt trigger 1 as the source. Instead of being incremented at a fixed rate, the clock is incremented by the occurrence of another external event on Schmitt trigger 1.

The syntax for calling the procedure is as follows:

```
READ_COUNTS_WAIT ( buffer, number, source, base, start, kw_desc, state );
```

Parameter	Type	Description
VAR buffer	ARRAY[first..last: INTEGER] OF INTEGER	Buffer that the counter is to be copied to after each interrupt
number	INTEGER	Number of elements to be copied to the count array
source	KW_RATE	Value indicating the source of the counts
base	KW_BASE_TYPE	Value indicating the base for counting
start	KW_START_TYPE	Value indicating how the clock is to be started
VAR kw_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR state	UNSIGNED	Status code indicating success (ES$NOR=0) or type of error (see Section 8.5)

The data types KW_RATE, KW_BASE_TYPE, and KW_START_TYPE, from KWINC.PAS, follow:

```
TYPE
  kw_rate = (
    kwv_stop,              { stop the clock }
    kwv_1MHz,              { 1000000 Hz }
    kwv_100kHz,            {  100000 Hz }
    kwv_10kHz,             {   10000 Hz }
    kwv_1kHz,              {    1000 Hz }
    kwv_100Hz,             {     100 Hz }
    kwv_ST1,               { Schmitt Trigger 1 determines the
                             clock frequency }
    kwv_line );            { Line frequency 50/60 Hz }

   kw_base_type =
     ( rtc_continuous,     { Time of event }
       rtc_zero_base );    { Elapsed time since previous event }

   kw_start_type =
     ( immediate,          { The real-time clock is started
                             immediately }
       event );            { The real-time clock is started
                             by an external event on Schmitt
                             trigger 2 }
```

KW_RATE specifies the source of the counts. The value KWV_STOP is illegal; KWV_1MHZ, KWV_100KHZ, KWV_10KHZ, KWV_1KHZ, and KWV_100HZ specify clock ticks at the respective rates; KWV_ST1 specifies counts of events logged on Schmitt trigger 1; and KWV_LINE specifies clock ticks at the line frequency (50 or 60 Hz).

KW_BASE_TYPE specifies the base for counting. The value RTC_CONTINUOUS specifies that the count at any event is continuous from the first event; RTC_ZERO_BASE specifies that the count resets to 0 after each event.

KW_START_TYPE specifies how the clock is to be started. The value IMMEDIATE specifies that the clock is to be started immediately; EVENT specifies that the clock is to be triggered by an external event on Schmitt trigger 2.

8.3.2 READ_COUNTS_SIGNAL

The READ_COUNTS_SIGNAL procedure reads a block of counts from the programmable real-time clock. READ_COUNTS_SIGNAL is identical to READ_COUNTS_WAIT, except that it returns immediately after issuing a read request to the KW driver. When the request is completed, the specified semaphore is signaled. (Correspondingly, READ_COUNTS_SIGNAL takes a reply parameter where READ_COUNTS_WAIT takes a state parameter.)

READ_COUNTS_SIGNAL can be used to record the time of external events or the time between external events. Also, two events can be monitored with respect to each other. See the READ_COUNTS_WAIT discussion of those operations.

The syntax for calling the procedure is as follows:

```
READ_COUNTS_SIGNAL ( buffer, number, source, base, start, kw_desc, reply );
```

Parameter	Type	Description
VAR buffer	ARRAY[first..last: INTEGER OF INTEGER	Buffer that the counter is to be copied to after each interrupt
number	INTEGER	Number of elements to be copied to the count array
source	KW_RATE	Value indicating the source of the counts
base	KW_BASE_TYPE	Value indicating the base for counting
start	KW_START_TYPE	Value indicating how the clock is to be started
VAR kw_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The data types KW_RATE, KW_BASE_TYPE, and KW_START_TYPE, from KWINC.PAS, follow:

```
TYPE
  kw_rate = (
    kwv_stop,             { stop the clock }
    kwv_1MHz,             { 1000000 Hz }
    kwv_100kHz,           {  100000 Hz }
    kwv_10kHz,            {   10000 Hz }
    kwv_1kHz,             {    1000 Hz }
    kwv_100Hz,            {     100 Hz }
    kwv_ST1,              { Schmitt Trigger 1 determines the
                            clock frequency }
    kwv_line );           { Line frequency 50/60 Hz }

  kw_base_type =
    ( rtc_continuous,     { Time of event }
      rtc_zero_base );    { Elapsed time since previous event }

  kw_start_type =
    ( immediate,          { The real-time clock is started
                            immediately }
      event );            { The real-time clock is started
                            by an external event on Schmitt
                            trigger 2 }
```

KW_RATE specifies the source of the counts. The value KWV_STOP is illegal; KWV_1MHZ, KWV_100KHZ, KWV_10KHZ, KWV_1KHZ, and KWV_100HZ specify clock ticks at the respective rates; KWV_ST1 specifies counts of events logged on Schmitt trigger 1; and KWV_LINE specifies clock ticks at the line frequency (50 or 60 Hz).

KW_BASE_TYPE specifies the base for counting. The value RTC_CONTINUOUS specifies that the count at any event is continuous from the first event; RTC_ZERO_BASE specifies that the count resets to 0 after each event.

KW_START_TYPE specifies how the clock is to be started. The value IMMEDIATE specifies that the clock is to be started immediately; EVENT specifies that the clock is to be triggered by an external event on Schmitt trigger 2.

Indication of success or failure of the read operation is returned in the status field of the KW driver reply packet.

8.3.3 START_RTCLOCK

The START_RTCLOCK procedure sets up the real-time clock for interval timing or as a free-running clock used to initiate A/D conversions. START_RTCLOCK starts the real-time clock running at a specified rate for either a single interval or a repeated interval. If signaling is specified, the binary or counting semaphore designated by the user will be signaled at the end of each interval. If no signaling is specified, clock interrupts are disabled; A/D conversions can be triggered at the end of each interval while the clock runs freely.

The packet-level equivalent of START_RTCLOCK is the IF$ENA function.

The syntax for calling the procedure is as follows:

```
START_RTCLOCK ( source, counts, single, start, signals, timer, kw_desc, state );
```

Parameter	Type	Description
source	KW_RATE	A value indicating the source of the counts
counts	INTEGER	Number of clock ticks at the specified rate (KW_RATE) until the counter overflows
single	BOOLEAN	Value indicating whether the clock is to run for a single interval (TRUE) or continuously at the specified rate (FALSE)
start	KW_START_TYPE	Value indicating how the clock is to be started
signals	BOOLEAN	Value indicating whether a semaphore is to be signaled after each interval (TRUE) or not (FALSE); if the value FALSE is specified, the clock is started with clock interrupts disabled and no signal is issued; FALSE should be specified when the clock is being used to initiate A/D conversions and no signaling is desired
VAR timer	STRUCTURE_DESC	Optional descriptor for the binary or counting semaphore to be signaled at the end of each interval
VAR kw_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR state	UNSIGNED	Status code indicating success (ES$NOR=0) or type of error (see Section 8.5)

The data types KW_RATE and KW_START_TYPE, from KWINC.PAS, are shown below:

```
TYPE
  kw_rate = (
    kwv_stop,           { stop the clock }
    kwv_1MHz,           { 1000000 Hz }
    kwv_100kHz,         {  100000 Hz }
    kwv_10kHz,          {   10000 Hz }
    kwv_1kHz,           {    1000 Hz }
    kwv_100Hz,          {     100 Hz }
    kwv_ST1,            { Schmitt Trigger 1 determines the
                          clock frequency }
    kwv_line );         { Line frequency 50/60 Hz }

  kw_start_type =
    ( immediate,        { The real-time clock is started
                          immediately }
      event );          { The real-time clock is started
                          by an external event on Schmitt
                          trigger 2 }
```

KW_RATE specifies the source of the counts. The value KWV_STOP is illegal; KWV_1MHZ, KWV_100KHZ, KWV_10KHZ, KWV_1KHZ, and KWV_100HZ specify clock ticks at the respective rates; KWV_ST1 specifies counts of events logged on Schmitt trigger 1; and KWV_LINE specifies clock ticks at the line frequency (50 or 60 Hz).

KW_START_TYPE specifies how the clock is to be started. The value IMMEDIATE specifies that the clock is to be started immediately; EVENT specifies that the clock is to be triggered by an external event on Schmitt trigger 2.

8.3.4 STOP_RTCLOCK

The STOP_RTCLOCK procedure stops the KWV11-C programmable real-time clock by disabling interrupts on the device.

The packet-level equivalent of STOP_RTCLOCK is the IF$DSA function.

The syntax for calling the procedure is as follows:

```
STOP_RTCLOCK ( kw_desc );
```

Parameter	Type	Description
VAR kw_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor

If a READ_COUNTS_SIGNAL request is in progress when this procedure is invoked, the KW driver reply to that request will indicate that the clock was stopped.

8.4 Request/Reply Packet Interface

The following packet-level functions provided by the KW driver are listed by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical
IF$GET (7)	Get Characteristics
IF$ENA (8)	Enable Clock
IF$DSA (9)	Disable Clock

If a request is received for an Open (IF$LOK or IF$ENT), the driver returns an illegal function status code (ES$IFN), which the ACP interprets as indicating that no device-dependent processing was required for that operation. However, as noted in Section 8.2, DIGITAL recommends that you do not perform file-oriented operations on a real-time clock.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

A single function modifier is recognized by the KW driver, as shown below:

Code	Function
FM$BSM (bit 13)	Signal binary/counting semaphore

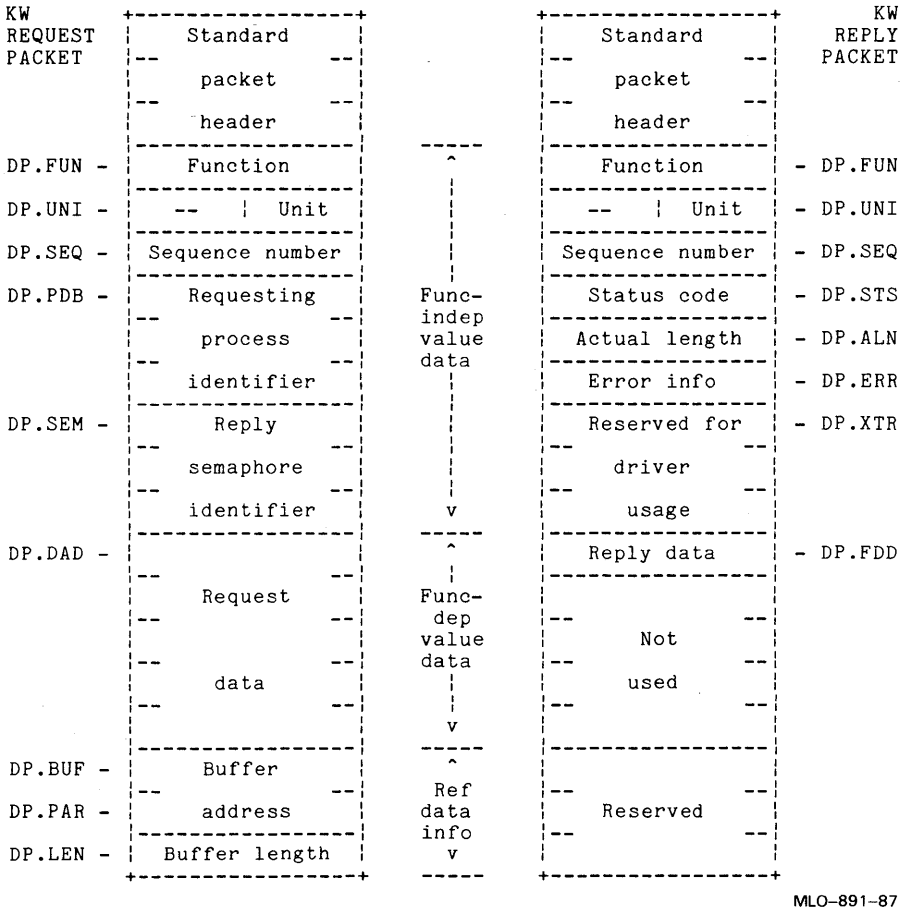
The KW driver consists of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each configured controller. I/O requests for a controller are sent (using a Pascal SEND or a MACRO–11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The request queue names and number of supported units for KW driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
Real-time clock	$KWc	1	0 (normally)

The letter c in a queue name represents a controller designation (A, B, ..., as specified in the KW driver prefix file).

The general format of the KW request and reply packets is shown below:



The function-independent portions of the packets shown above are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit number for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of KW driver function.

Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

8.4.1 Read Physical Function

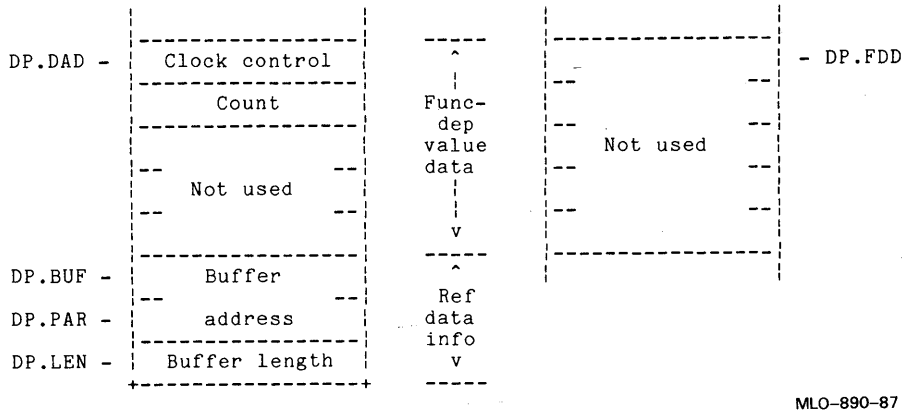
The Read Physical (IF$RDP) function reads a block of counts from the programmable real-time clock. It can be used to record the time of external events or the time between external events. Also, two events can be monitored with respect to each other.

External events are detected by Schmitt triggers, two of which reside on the KWV11–C board. The primary Schmitt trigger is the second one, ST2. It can be used to cause interrupts or to start the clock. The first Schmitt trigger, ST1, can be used only to increment the clock.

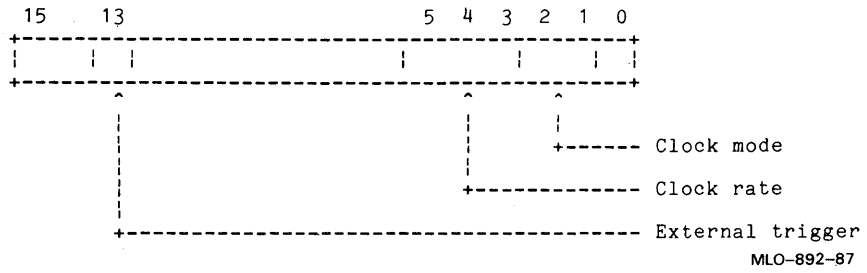
To record the time of external events or the time between external events, a fixed rate is specified for the source. The clock is started by either the driver setting the GO bit in the CSR or by the first external event on Schmitt trigger 2. When the clock starts, the clock counter is cleared and subsequently incremented at the specified rate. When an event occurs on Schmitt trigger 2, the value of the counter is transferred to the buffer/preset register, and an interrupt is requested. If zero-base is specified, the counter is zeroed; otherwise, the clock is incremented from its current value. (Continuous incrementing gives the time of the event, and zero-base gives the time between events.) The ISR reads the count from the buffer/preset register and copies it to the user-specified buffer. An overrun condition occurs when a second external event occurs before the ISR has read the count from the preceding external event and indicates that the events are occurring too quickly for the system to handle. An overflow condition occurs when the clock counter overflows before a second external event occurs and may indicate that too high a rate was specified.

To measure the relative frequency of one event to another, proceed as above, but specify Schmitt trigger 1 as the source. Instead of being incremented at a fixed rate, the clock is incremented by the occurrence of another external event on Schmitt trigger 1.

The function-dependent portions of the read request and reply packets are shown below:



The clock control word has the format shown below:



The bit fields shown contain clock control information. Proceeding from right to left in the format above:

- Bits 1 and 2 specify the clock's mode of operation, as shown below:

Value	Mode
2	External event timing
3	External event timing from zero base

In external-event timing mode, you can generate a pulse train while monitoring external events, record the time of external events, or count external events. Two external events can be monitored with respect to each other. The counter increments at the user-selected clock rate or at the rate of external input until it overflows. An input at Schmitt trigger 2 (ST2) causes the contents of the counter to be loaded into the buffer/preset register (BPR), where it can be read by the KW device driver.

External-event timing from a zero base is the same as external-event timing, except that the clock is reset to 0 after each event.

- Bits 3 through 5 select the clock rate, as shown below:

Value	Rate of Operation
0	STOP
1	1 MHz
2	100 kHz
3	10 kHz
4	1 kHz
5	100 Hz
6	ST1 external input
7	Line (50/60 Hz)

- Bit 13, if set, specifies that the clock is to be started by an external event (Schmitt trigger 2); otherwise, the KW driver starts the clock immediately.

The count word (offset DP.DAD+2) supplies one of the following values:

- The number of clock pulses that will generate the time delay required at the user-selected clock frequency
- The number of line inputs (BEVNT) that will generate a real-time reference to record the time of an external event at Schmitt trigger 2 (ST2)
- The number of external events to be counted at Schmitt trigger 1 (ST1) before an overflow occurs

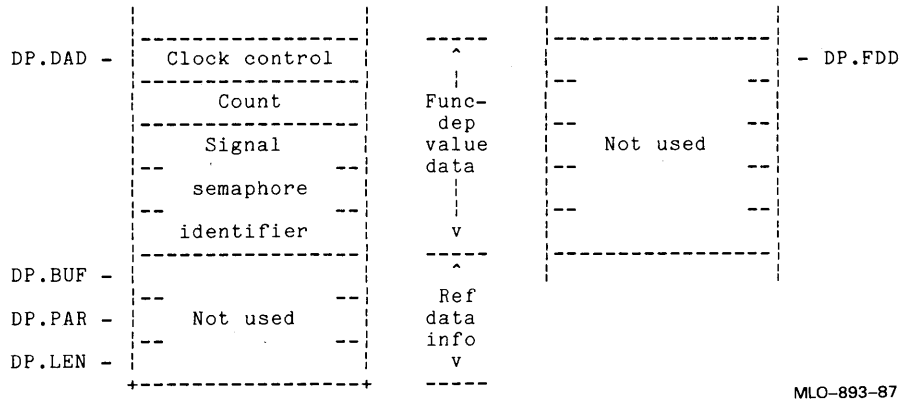
The KW device driver copies the two's complement of the count value to the clock's buffer/preset register.

The buffer-address and buffer-length fields specify the buffer to which the counter is to be copied after each interrupt. The buffer address must be on a word boundary.

8.4.2 Enable Clock Function

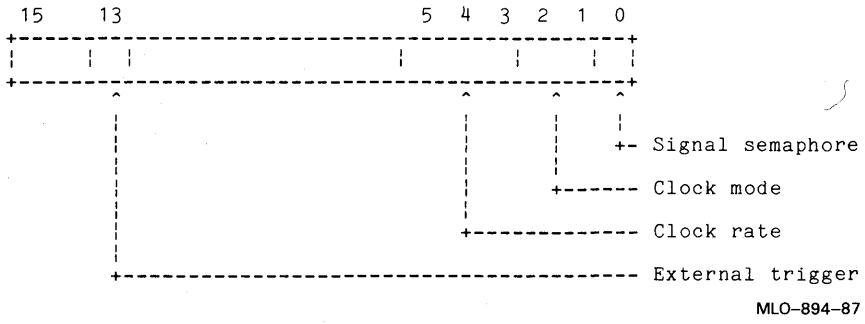
The Enable Clock (IF$ENA) function sets up the real-time clock for interval timing or as a free-running clock used to initiate A/D conversions. Enable Clock starts the real-time clock running at a specified rate for either a single interval or a repeated interval. If signaling is specified, the binary or counting semaphore designated by the user will be signaled at the end of each interval. If no signaling is specified, clock interrupts are disabled; A/D conversions can be triggered at the end of each interval while the clock runs freely.

The function-dependent portions of the Enable Clock request and reply packets are shown below:



MLO-893-87

The clock control word has the format shown below:



MLO-894-87

The bit fields shown contain clock control information. Proceeding from right to left:

- Bit 0, if set, causes a binary or counting semaphore to be signaled after each clock interrupt; the bit should not be set when the clock is being used to trigger A/D conversions.
- Bits 1 and 2 specify the clock's mode of operation, as shown below:

Value	Mode
0	Single interval
1	Repeated interval

In single-interval mode, the clock's counter is set, and the clock increments at the user-selected clock rate until it overflows and stops.

In repeated-interval mode, the clock's counter is set, and the clock increments at the user-selected clock rate until it overflows. Upon overflow, the clock-overflow signal is generated, the clock's counter is reset, and counting continues. This mode is used for repeated clock signals.

- Bits 3 through 5 select the clock rate, as shown below:

Value	Rate of Operation
0	STOP
1	1 MHz
2	100 kHz
3	10 kHz
4	1 kHz
5	100 Hz
6	ST1 external input
7	Line (50/60 Hz)

- Bit 13, if set, specifies that the clock is to be started by an external event (Schmitt trigger 2); otherwise, the KW driver starts the clock immediately

The count word (offset DP.DAD+2) supplies one of the following values:

- The number of clock pulses that will generate the time delay required at the user-selected clock frequency
- The number of line inputs (BEVNT) that will generate a real-time reference to record the time of an external event at Schmitt trigger 2 (ST2)
- The number of external events to be counted at Schmitt trigger 1 (ST1) before an overflow occurs

The KW device driver copies the two's complement of the count value to the clock's buffer/preset register.

The signal semaphore identifier field (beginning at offset DP.DAD+4) specifies the binary or counting semaphore, if any, to be signaled after each clock interrupt.

8.4.3 Disable Clock Function

The Disable Clock (IF$DSA) function stops the KWV11–C programmable real-time clock by disabling interrupts on the device.

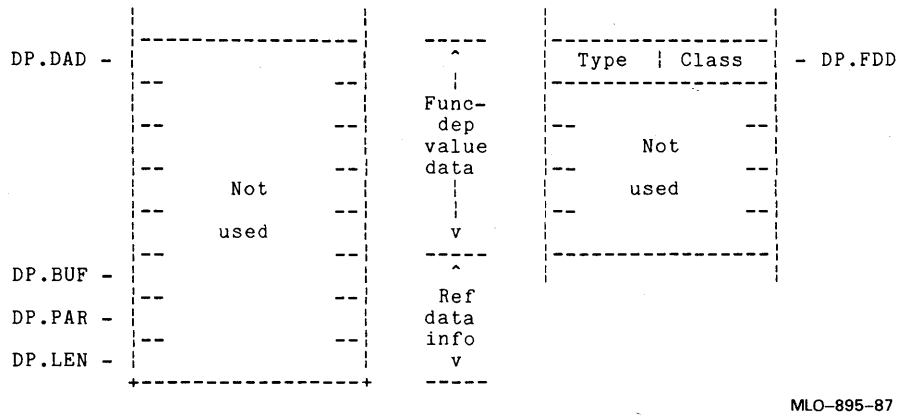
If a read request is in progress when the clock is disabled, the KW driver reply to that request will indicate that the clock was stopped.

The function-dependent portions of the Disable Clock request and reply packets are not used.

8.4.4 Get Characteristics Function

The KW Get Characteristics (IF$GET) function returns the codes for real-time clock device class and type in the function-dependent portion of the reply message.

The function-dependent portions of the Get Characteristics request and reply packets are shown below:



In the preceding information:

- Class is DC$RLT for real-time device class.
- Type is RT$KWV for the KWV11-C.

8.5 Status Codes

If an error is detected during an I/O operation by the real-time clock or the KW driver, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If no error is detected during the I/O operation, a value of ES$NOR (0) is returned.

The KW driver returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O request aborted by user
ES$IVM	HARD_IO	Invalid mode
ES$IFN	SOFT_IO	Illegal function

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for kernel-detected errors that the KW driver raises rather than passing back to the requesting process.

8.6 KW Driver Prefix File

Figure 8-1 shows the KW driver prefix module. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The symbols KW$IPR, KW$PPR, and KW$HPR define the initialization and request-handling software priorities for the driver process and the hardware interrupt priority for the controller(s).

The DRVCF$ macro contains a field for the number of controllers on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced by the driver. It gives the controller name, number of units (1), CSR and vector addresses, and unit number.

Note that the interrupt vectors must also be specified in the system configuration file, using the DEVICES macro.

Figure 8–1: KW Driver Prefix File (KWPFX.MAC)

```
.title   KWPFX   - Real-Time Clock Prefix Module

;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1985, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS
; RESERVED.
;

    .mcall   drvcf$
    .mcall   ctrcf$
    .mcall   kwisz$

    kwisz$

KW$PPR   ==   184.       ; Process priority
KW$HPR   ==   5          ; Hardware priority
KW$IPR   ==   250.       ; Process initialization priority

         drvcf$   dname=KW,nctrl=1
         ctrcf$   cname=A,nunits=1.,csrvec=<170420,440>,units=<0:0>
;        ctrcf$   cname=B,nunits=1.,csrvec=<170420,410>,units=<1:1>

   .end
```

Chapter 9
Peripheral Processor DMA Driver

This chapter describes the use of the MicroPower/Pascal peripheral processor DMA (QD) driver, which supports I/O operations on the KXT11–CA/KXJ11–CA 2-channel DMA transfer controller (DTC). The KXT11–CA/KXJ11–CA DTC transfers data between any two of the following:

- A local address
- A Q-bus address
- An I/O device on the arbiter processor's Q-bus
- An I/O device connected directly to the KXT11–CA or KXJ11–CA

Note

The QD driver can be used to perform DMA transfers via a KXT11–CA or KXJ11–CA parallel port. The QD driver coordinates such transfers with the KXT11–CA/KXJ11–CA parallel I/O (YK) driver. For details, see Section 9.3.4.

9.1 QD Driver Features and Capabilities

The QD driver supports DMA read and write operations, channel allocation and deallocation, and the returning of device status information, as follows:

- Read and write operations transfer a specified number of data bytes between two locations. Data is transferred by word (the default) or byte, using direct memory access. Once the operation is initiated, there is no processor intervention.
- Read and write operations have pattern recognition capabilities for terminating a transfer when a specified pattern is found or a search limit is reached.
- Channel allocation dedicates a specified unit for the exclusive use of the calling process.
- Get Characteristics operations report standard device characteristics and return the contents of device registers.

9.2 Performing KXT11-CA/KXJ11-CA DMA I/O

For most MicroPower/Pascal KXT11-CA or KXJ11-CA applications, you perform DMA transfers by invoking Pascal support routines—$DMA_TRANSFER, $DMA_SEARCH, and so forth. Those routines provide high-level nonfile access to the KXT11-CA/KXJ11-CA DTC channels. The QD support routines issue Pascal SEND requests to the request queue semaphore of the QD driver. The routines are described in Section 9.3.

Note

You cannot perform file-oriented Pascal operations on the KXT11-CA/KXJ11-CA DTC. If you try to open a KXT11-CA/KXJ11-CA DTC file, the QD driver returns an unsupported function (ES$UFN) exception code, and the OTS raises the exception (unless you requested a status return).

In addition to invoking the Pascal support routines, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the DTC interrupt vector addresses
2. Edit the KXT11-CA/KXJ11-CA DMA driver prefix file to reflect:
   - Number of controllers (normally 1)
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, number of controller units and their identifying numbers (0, 1)
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - KXT11-CA/KXJ11-CA DMA driver process
   - Pascal KXT11-CA/KXJ11-CA DTC support routines (from kit files DMA.PAS and QDINC.PAS)

For more information on setting up your application software for KXT11-CA/KXJ11-CA DMA I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 9.6 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

As an alternative to using the Pascal support routines for KXT11-CA/KXJ11-CA DMA I/O, you can issue your own Pascal or MACRO-11 packet-level requests to the driver (low-level nonfile access). In such a case, do not build the support routines into your application.

The following sections describe the Pascal support routine interface to the QD driver, the lower-level request/reply packet interface, the status codes that can be returned to users of either interface, and the QD driver prefix file.

9.3 Pascal Support Routine Interface

The following support routines, written in Pascal and independent of the file system, provide a high-level interface to the KXT11-CA/KXJ11-CA DMA channels:

- $DMA_TRANSFER function
- $DMA_SEARCH function
- $DMA_SEARCH_TRANSFER function
- $DMA_ALLOCATE function
- $DMA_DEALLOCATE functio
- $DMA_GET_STATUS function

The following sections describe the Pascal functions for KXT11-CA/KXJ11-CA DMA I/O. Each function takes an I/O packet, fills it with information based on the function parameters, and sends the packet to the QD driver.

If a reply semaphore is provided in the function call, the function returns immediately after sending the driver request. When the operation is complete, the driver sends a standard device driver reply via the specified semaphore. (The driver reply is described in Section 9.4.) The completion status returned in the reply packet must be processed by a routine that is waiting on the semaphore. For transfer or search operations, the routine that waits on the semaphore must also process the actual-length information in the packet.

If no reply semaphore is provided—or if the function has no reply parameter ($DMA_ALLOCATE or $DMA_DEALLOCATE)—the function waits for the driver reply before returning to the caller.

The following files on the MicroPower/Pascal distribution kit are required for using the functions:

Name	Description
IOPKTS.PAS	Pascal I/O include file
DMA.PAS	KXT11-CA/KXJ11-CA DMA function source module
QDINC.PAS	KXT11-CA/KXJ11-CA DMA function include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

The following data type from QDINC.PAS, referenced throughout this section, defines the QD unit numbers for the support routine interface:

```
TYPE

DMA$UNIT_NUMBER = 0..1; { 0 for channel A, 1 for channel B }
```

9.3.1 $DMA_TRANSFER

The $DMA_TRANSFER function transfers data between two locations. Each start address is specified as part of a record of type DMA$ADDRESS. Each address can specify a Q-bus memory address, an I/O device on the arbiter's Q-bus, a local memory address, or an I/O device connected directly to the KXT11-CA or KXJ11-CA. In addition, each address record contains control bit settings that select such options as I/O page reference and Q-bus mapping, address incrementation, hardware request synchronization (used for DMA transfer via a KXT11-CA/KXJ11-CA parallel port), and use of byte-mode transfers.

The function returns a value of type DMA$BYTE_COUNT—the count of bytes transferred (0 if an error occurred or a reply parameter was provided).

The syntax for calling the function is as follows:

```
$DMA_TRANSFER ( source, dest, count, unit, reply )
```

Parameter	Type	Description
source	DMA$ADDRESS	Q-bus or local address from which data will be transferred
dest	DMA$ADDRESS	Q-bus or local address to which data will be transferred
count	DMA$BYTE_COUNT	Number of bytes to transfer
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)
reply	DMA$SEM_POINTER	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL

The data types DMA$ADDRESS, DMA$BYTE_COUNT, and DMA$SEM_POINTER, from QDINC.PAS, are shown below.

Note

The DMA$ADDRESS data structure and its addressing control bits are diagrammed in Section 9.4.1.

```
TYPE
  DMA$ADDR_SPACE = (DMA$IBUS, DMA$QBUS);     { local or qbus space }
  DMA$INCR_OPTION = (DMA$UP, DMA$DOWN, DMA$NOINC); { increment up,
                                                down, or not at all }
  DMA$WAIT_OPTION = (DMA$WAIT_0)             { only 0 wait states is
                                               supported}
  DMA$REQ_OPTION = (DMA$NOWFR, DMA$WFR);     { wait for request line active}
  DMA$IO_OPTION = (DMA$NOIO, DMA$IO);        {   access I/O page addresses }
  DMA$BYTE_OPTION = (DMA$NOBYTE, DMA$BYTE); { nobyte (word) mode or byte
                                               mode }
```

```
DMA$ADRTYP_OPTION = (DMA$VIRTUAL, DMA$PHYSICAL); { virtual - need to
                                                   convert   to  physical,
                                                   physical  -  no  need  to
                                                   convert }

DMA$ADDRESS = PACKED RECORD                        { source or destination
                                                   dma address }
  LOW: [POS(00),WORD] UNSIGNED;                    { low portion of 22-bit
                                                   address}

  HIGH: [POS(16),BIT(6)] 0..63;                    { high portion of 22-bit
                                                   address}
   ADRTYP: [POS(22),BIT(1) DMA$ADRTYP_OPTION { physical or virtual }
   IO: [POS(24),BIT(1)] DMA$IO_OPTION;      { IO mode ? }
   WS: [POS(25),BIT(2)] DMA$WAIT_OPTION;    { number of wait states to add }
   INC: [POS(27),BIT(2)] DMA$INCR_OPTION;   { UP or DOWN or NOINC }
   WFR: [POS(29),BIT(1)] DMA$REQ_OPTION;    { wait for request ? }
   BM: [POS(30),BIT(1)] DMA$BYTE_OPTION;    { byte mode ? }
   SPACE: [POS(31),BIT(1)] DMA$ADDR_SPACE   { IBUS OR QBUS }
END;

DMA$BYTE_COUNT = UNSIGNED;                         { number of bytes to transfer }

DMA$SEM_POINTER = ^ SEMAPHORE_DESC;                { pointer to sem desc }


CONST
{
These constants are used to initialize variables of
type DMA$ADDRESS.
}

DMA$NORM_IBUS_ADDRESS = DMA$ADDRESS ( 0,  0,  DMA$VIRTUAL, DMA$NOIO,
                                      DMA$WAIT_0, DMA$UP, DMA$NOWFR,
                                      DMA$NOBYTE, DMA$IBUS );
DMA$NORM_QBUS_ADDRESS = DMA$ADDRESS ( 0,  0,  DMA$PHYSICAL DMA$NOIO,
                                      DMA$WAIT_0, DMA$UP, DMA$NOWFR,
                                      DMA$NOBYTE, DMA$QBUS );
```

In the function call, the source and the destination arguments must be records of type DMA$ADDRESS.

If the DMA$VIRTUAL option is specified, either the interface routine or the driver converts the address to a physical address. This is necessary because the DTC requires physical addresses. If the DMA$PHYSICAL option is specified, the address is not converted. You must specify a physical address for any Q-bus address. For unmapped applications, virtual addresses are the same as physical addresses. Therefore, you can use either option. There is a little more overhead if you use the DMA$VIRTUAL option, but the same source code can then be used in both mapped and unmapped applications.

If you wish to convert a virtual buffer address to a physical address yourself, instead of letting the interface routine or the driver do it, you can use the function $TRAN_VIRT_PHYS in DMA.PAS to perform the conversion. This is necessary if a mapped arbiter application wants to tell a KXT11-CA or KXJ11-CA peripheral processor to use the DTC to transfer to or from a Q-bus buffer. The arbiter application must convert the virtual Q-bus buffer address to a physical Q-bus address and pass that physical address to the KXT11-CA or KXJ11-CA.

If the addressing control option DMA$IO is specified, the DMA address references the I/O page rather than normal memory locations.

The DMA$WAIT_n options specify the number of wait states to be programmed into DMA address access. Only a value of 0 wait states is supported.

The DMAUP, DMADOWN, and DMA$NOINC options cause the address to be incremented, decremented, or held constant during a transfer.

If the DMA$WFR option is specified, the driver waits for a hardware request (request line active) for a transfer. This option is used to coordinate the DTC and the KXT11-CA/KXJ11-CA PIO port controller for a DMA transfer via a PIO port. (See Section 9.3.4.) Since this option applies to the transfer in general and not to just one address, it may be specified in either the source or the destination address record.

If the DMA$BYTE option is specified, data is transferred in byte mode rather than word mode. Byte mode is supported only for I/O to or from a local (internal bus) byte-oriented device—for example, a parallel port or an asynchronous line—with the other address even. (Section 9.3.4 describes DMA transfer via a parallel port.) Use word mode for memory-to-memory (and Q-bus) transfers. Byte-to-word and word-to-byte funneling are not supported. Since this option applies to the transfer in general and not to just one address, it may be specified in either the source or the destination address record.

If the DMA$QBUS option is specified, the DMA address is mapped to Q-bus space rather than the internal bus. (Byte mode cannot be selected when DMA$QBUS is specified.)

9.3.2 $DMA_SEARCH

The $DMA_SEARCH function searches data for a user-specified search value, beginning at a user-specified Q-bus or local address. The search terminates either when the search value is matched or when a specified byte count expires.

The function returns a value of type DMA$BYTE_COUNT—the number of bytes searched (0 if an error occurred or if a reply parameter was provided).

The syntax for calling the function is as follows:

```
$DMA_SEARCH ( source, count, val, mask, unit, reply )
```

Parameter	Type	Description
source	DMA$ADDRESS	Q-bus or local address at which search will begin
count	DMA$BYTE_COUNT	Maximum number of bytes to search
val	UNSIGNED	Search value
mask	UNSIGNED	Optional search mask; default is 0
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)
reply	DMA$SEM_POINTER	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL

The DMA$ADDRESS data type is listed and described in Section 9.3.1 and is diagrammed in Section 9.4.1.

The DMA$BYTE_COUNT and DMA$SEM_POINTER data types, from QDINC.PAS, are shown below:

```
TYPE
  DMA$BYTE_COUNT = UNSIGNED;    { number of bytes to transfer or search }
  DMA$SEM_POINTER = ^ SEMAPHORE_DESC;    { pointer to sem desc }
```

Bits set to 1 in the mask parameter mask out bits in the object word. For example, to search only the low-order byte of each word in a buffer, you should specify a mask parameter with all eight high-order bits set to 1. Thus, the low-order byte of each word in the buffer will be compared with the low-order byte of the search value parameter.

To search for a byte in a buffer, you must perform two search operations. You must first search the low-order byte of each word and mask out the high, then search the high-order byte of each word and mask out the low. When the high-order byte is being searched, the search value must be shifted to the high-order byte. For example, to search a buffer for the byte value, value_to_find, the appropriate search value and search mask parameters would be, for the low-order search, VAL := value_to_find and MASK := %O'177400', and, for the high-order search, VAL := (value_to_find * 256) and MASK := %O'377'.

9.3.3 $DMA_SEARCH_TRANSFER

The $DMA_SEARCH_TRANSFER function causes data to be transferred until either a search value is matched or a byte count expires. Each start address is specified with a record of type DMA$ADDRESS. Each address can specify a Q-bus memory address, an I/O device on the arbiter's Q-bus, a local memory address, or an I/O device connected directly to the KXT11-CA or KXJ11-CA. In addition, each address record contains control bit settings that select such options as I/O page reference and Q-bus mapping, address incrementation, hardware request synchronization (used for DMA transfer via a KXT11-CA/KXJ11-CA parallel port), and use of byte mode transfers.

The function returns a value of type DMA$BYTE_COUNT—the number of bytes transferred (0 if an error occurred or if a reply parameter was provided).

The syntax for calling the function is as follows:

```
$DMA_SEARCH_TRANSFER ( source, dest, count, val, mask, unit, reply )
```

Parameter	Type	Description
source	DMA$ADDRESS	Q-bus or local address at which search will begin and from which data will be transferred
dest	DMA$ADDRESS	Q-bus or local address to which data will be transferred
count	DMA$BYTE_COUNT	Maximum number of bytes to transfer
val	UNSIGNED	Search value
mask	UNSIGNED	Optional search mask; default is 0
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)
reply	DMA$SEM_POINTER	Optional pointer to initialized reply queue semaphore descriptor; default is NIL

The DMA$ADDRESS data type is listed and described in Section 9.3.1 and is diagrammed in Section 9.4.1.

The DMA$BYTE_COUNT and DMA$SEM_POINTER data types, from QDINC.PAS, are shown below:

```
TYPE
  DMA$BYTE_COUNT = UNSIGNED;          { number of bytes to transfer or search }
  DMA$SEM_POINTER = ^ SEMAPHORE_DESC;  { pointer to sem desc }
```

In the function call, the source and the destination arguments must be records of type DMA$ADDRESS.

9.3.4 KXT11-CA/KXJ11-CA PIO with DMA

If you want to perform DMA transfers via a KXT11-CA or KXJ11-CA parallel port, you must first set up a DMA Read or a DMA Write request packet and send it to the YK driver and wait for the reply. If the reply indicates normal status, you then send a DMA transfer command to the DMA (QD) driver; otherwise, you report an error or wait. You must wait for each request to complete, since only one PIO DMA operation can be in progress at a time. After the DMA transfer completes, you send a DMA Complete request to the YK driver, which unlocks the queue of requests for that port.

For guidelines to follow when performing DMA I/O on a KXT11-CA/KXJ11-CA parallel port—and a sample program—see Section 6.4.2.4.

9.3.5 KXT11-CA/KXJ11-CA I/O Using SLU2A or SLU2B with DMA

You can use the DMA controller to transfer data to or from either of the serial line ports SLU2 channel A and SLU2 channel B. User-supplied software must perform the necessary setup to make this work, however, since the TT and XS drivers do not support this capability.

9.3.6 $DMA_GET_STATUS

The $DMA_GET_STATUS function returns status information—the contents of device registers—from the specified DTC channel into a user-supplied buffer. The function returns a Boolean value indicating success (TRUE) or failure (FALSE).

The syntax for calling the function is as follows:

```
$DMA_GET_STATUS ( unit, regbuf, regbuf_size, reply )
```

Parameter	Type	Description
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)
VAR regbuf	DMA$DEVICE_REGS	Buffer to which status information is to be returned
regbuf_size	INTEGER	Optional buffer size in bytes; default is DMA$REGBUF_SIZE (92)
reply	DMA$SEM_POINTER	Optional pointer to an initialized reply queue semaphore descriptor; default is NIL

The data types DMA$SEM_POINTER and DMA$DEVICE_REGS, from QDINC.PAS, are shown below.

Note

The DMA$DEVICE_REGS data structure is diagrammed in Section 9.4.3.

```
TYPE
  DMA$SEM_POINTER = ^ SEMAPHORE_DESC;   { pointer to sem desc }

  DMA$DEVICE_REGS = PACKED RECORD   { Device registers }
    caoff_b_1 : unsigned;
    caoff_b_0 : unsigned;
    baoff_b_1 : unsigned;
    baoff_b_0 : unsigned;
    caoff_a_1 : unsigned;
    caoff_a_0 : unsigned;
    baoff_a_1 : unsigned;
    baoff_a_0 : unsigned;
```

```
    catag_b_1 : unsigned;
    catag_b_0 : unsigned;
    batag_b_1 : unsigned;
    batag_b_0 : unsigned;
    catag_a_1 : unsigned;
    catag_a_0 : unsigned;
    batag_a_1 : unsigned;
    batag_a_0 : unsigned;
    chaino_1 : unsigned;
    chaino_0 : unsigned;
    chaint_1 : unsigned;
    chaint_0 : unsigned;
    isr_1 : unsigned;
    isr_0 : unsigned;
    stat_1 : unsigned;
    stat_0 : unsigned;
    coc_1 : unsigned;
    coc_0 : unsigned;
    boc_1 : unsigned;
    boc_0 : unsigned;
    mmr : unsigned;
    junk : [WORD(7)];
    pat_1 : unsigned;
    pat_0 : unsigned;
    msk_1 : unsigned;
    msk_0 : unsigned;
    cmr_l_1 : unsigned;
    cmr_l_0 : unsigned;
    cmr_h_1 : unsigned;
    cmr_h_0 : unsigned;
    inv_1 : unsigned;
    inv_0 : unsigned;
  END;

CONST

{ Constant for the size of the register file }
  DMA$REGBUF_SIZE = 92;
```

If you specify the reply parameter, values for the device class and type are returned in a standard driver reply message. (See Section 9.4.3.)

If you specify a register buffer size of less than 92, only the number of bytes you request will be returned.

9.3.7 $DMA_ALLOCATE

The $DMA_ALLOCATE function allocates a specified DTC channel for the exclusive use of the calling process. This function returns a Boolean value indicating success (TRUE) or failure (FALSE).

The syntax for calling the function is as follows:

```
$DMA_ALLOCATE ( unit )
```

Parameter	Type	Description
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)

9.3.8 $DMA_DEALLOCATE

The $DMA_DEALLOCATE function reverses the effect of a previous DEALLOCATE > QD driver) $DMA_ALLOCATE call. This function returns a Boolean value indicating success (TRUE) or failure (FALSE).

The syntax for calling the function is as follows:

```
$DMA_DEALLOCATE ( unit )
```

Parameter	Type	Description
unit	DMA$UNIT_NUMBER	Optional unit number; default is 0 (channel A)

9.3.9 KXT11-CA/KXJ11-CA DMA Sample Program

Figure 9–1 shows a sample program, usable with the distributed QD driver prefix file, that demonstrates two operations:

- DMA transfers from a local buffer to the Q-bus and then back to another local buffer
- DMA search and transfer from one local buffer to another

Figure 9–1: KXT11–CA/KXJ11–CA DMA Sample Program

```
[ SYSTEM(MICROPOWER), PRIORITY(50),
  DATA_SPACE(2100), STACK_SIZE (400)] PROGRAM QDTST;   {need more stack
                                                        for $tran_virt_phys function
                                                        call in dma.pas}

{$NOLIST}
%include 'iopkts.pas'   {get common I/O definitions.}
%include 'escode.pas'   {get exception codes}
%include 'qdinc.pas'    {get the QD data structures
                           and interface}
{$LIST}
CONST
  BUFSIZE = %o'2000';  {byte size}
  QBUSBUF = DMA$ADDRESS ( %O'32000', %O'1', DMA$PHYSICAL, DMA$NOIO,
            DMA$WAIT_O, DMA$UP, DMA$NOWFR, DMA$NOBYTE,
            DMA$QBUS ); {address is 32000(8) in low 16
                 bits, 1 in high 6 bits = 232000(8)}
VAR
  buf1, buf2 :    packed array[1..bufsize] of BYTE;   {2 buffers}
  address_1, address_2 : DMA$ADDRESS;   {addresses for DMA calls}
  i,k : INTEGER;    {loop counters}
  error : INTEGER;    {error flag 0->success, 1-> failure}
  un : DMA$UNIT_NUMBER;  {unit or channel number for DMA device}
  my_reply_semaphore : SEMAPHORE_DESC;  {reply semaphore descriptor}
  my_reply_packet : IO_REPLY;   {reply semaphore}


BEGIN
      IF NOT CREATE_QUEUE_SEMAPHORE (DESC := my_reply_semaphore) THEN
        WRITELN ('Semaphore create failed');

      error := 0;       {clear error flag};
      un := 0;       {use unit 0};

{*
 * For this test, transfer to and from qbus, checking data.
 *}
      FOR i := 1 to BUFSIZE DO      { fill buffer with data }
        BEGIN
          buf1[i] := i mod 256;
          buf2[i] := 0;
        END;
      address_1 := DMA$NORM_IBUS_ADDRESS;
      address_1.low := (ADDRESS(BUF1))::UNSIGNED;
      address_2 := DMA$NORM_IBUS_ADDRESS;
      address_2.low := (ADDRESS(BUF2))::UNSIGNED;

      IF 0 = $DMA_TRANSFER (      { transfer... }
        UNIT := un,               { on this unit }
        SOURCE := address_1,      { from my local buffer }
        DEST := QBUSBUF,          { to the qbus buffer }
        COUNT := BUFSIZE )        { this much }
      THEN
    WRITELN ('Test 1 failed on write');
```

```
      IF 0 = $DMA_TRANSFER (        { transfer... }
        UNIT := un,                 { on this unit }
        DEST := address_2,          { to my local buffer }
        SOURCE := QBUSBUF,          { from the qbus buffer }
        COUNT := BUFSIZE )          { this much }
      THEN
    WRITELN ('Test 1 failed on read');

      FOR i := 1 to BUFSIZE DO              { check the data for errors }
        BEGIN
          IF ( buf1[i] <> buf2[i] ) THEN
       error := 1;
        END;

      IF ERROR=1 THEN WRITELN ('Test 1 data corrupted')
        ELSE WRITELN ('Test 1 passed');

      error := 0;   {clear error flag for next test}
      un := 1;      {use unit 1};

{*
 * For the next test, transfer ibus to ibus, searching.
 *}

      FOR i := 1 to BUFSIZE DO          { clear buf2 }
          buf2[i] := 0;

      address_1 := DMA$NORM_IBUS_ADDRESS;
      address_1.low := (ADDRESS(BUF1))::UNSIGNED;
      address_2 := DMA$NORM_IBUS_ADDRESS;
      address_2.low := (ADDRESS(BUF2))::UNSIGNED;

      k := $DMA_SEARCH_TRANSFER (       { transfer... }
        UNIT := un,                     { on this unit }
        SOURCE := address_1,            { from my local buffer }
        DEST := address_2,              { to my second buffer }
        VAL := 49,                      { looking for a 49 }
        MASK := %o'177400',             { in the low byte }
        COUNT := bufsize,               { this much }
        REPLY := address (my_reply_semaphore) );  {reply semaphore}

{WAIT FOR TRANSFER TO COMPLETE - Could do some work here first}

      RECEIVE (DESC := my_reply_semaphore,
          VAL_DATA := my_reply_packet,
          VAL_LENGTH := SIZE (my_reply_packet) );

        IF my_reply_packet.status <> es$nor THEN
        WRITELN ('Test 2 failed')
        ELSE
          BEGIN         {49th byte should match. Since in word mode,
                         byte count should be 50, for full word}
            IF (my_reply_packet.actual_length <> 50) THEN
              WRITELN ('Test 2 matched on wrong byte');
       FOR i := 1 TO my_reply_packet.actual_length DO
             IF ( buf1[i] <> buf2[i] ) THEN
          error := 1;
       IF error = 1 THEN
         WRITELN ('Test 2 data corrupted');
       IF ((my_reply_packet.actual_length=50) AND (error = 0)) THEN
         WRITELN ('Test 2 passed');
   END;

   END.
```

9.4 Request/Reply Packet Interface

The packet-level functions provided by the KXT11-CA/KXJ11-CA DMA driver are listed below by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical
IF$WTP (3)	Write Physical
IF$GET (7)	Get Characteristics
IF$ALL (8)	Allocate Channel
IF$DEA (9)	Deallocate Channel

If a request is received for an Open (IF$LOK or IF$ENT), the driver returns an unsupported function code (ES$UFN). That causes the OTS to raise the exception, provided that the OTS/ACP issued the open request and the user's OPEN statement did not specify a status return.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the QD driver are shown below by symbolic code and bit position:

Code	Function
FM$TTO (null)	Transfer data to/from DMA address without searching the data (default for QD read or write)
FM$TSO (bit 6)	Search data beginning at DMA address until either a search value is matched or a byte count (DP.SLN) expires; return count of bytes searched in actual-length field of reply packet (QD read or write)
FM$TTS (bit 7)	Transfer and search data until either a search value is matched or a byte count (DP.LEN) expires; return count of bytes transferred in actual-length field of reply packet (QD read or write)
FM$BSM (bit 13)	Signal binary/counting semaphore

The QD driver consists of an initialization process, which lowers its priority to become the DMA controller's request handler process.

Note

The QD driver supports multiple controllers, but normally just one controller is configured. See Section 9.6.

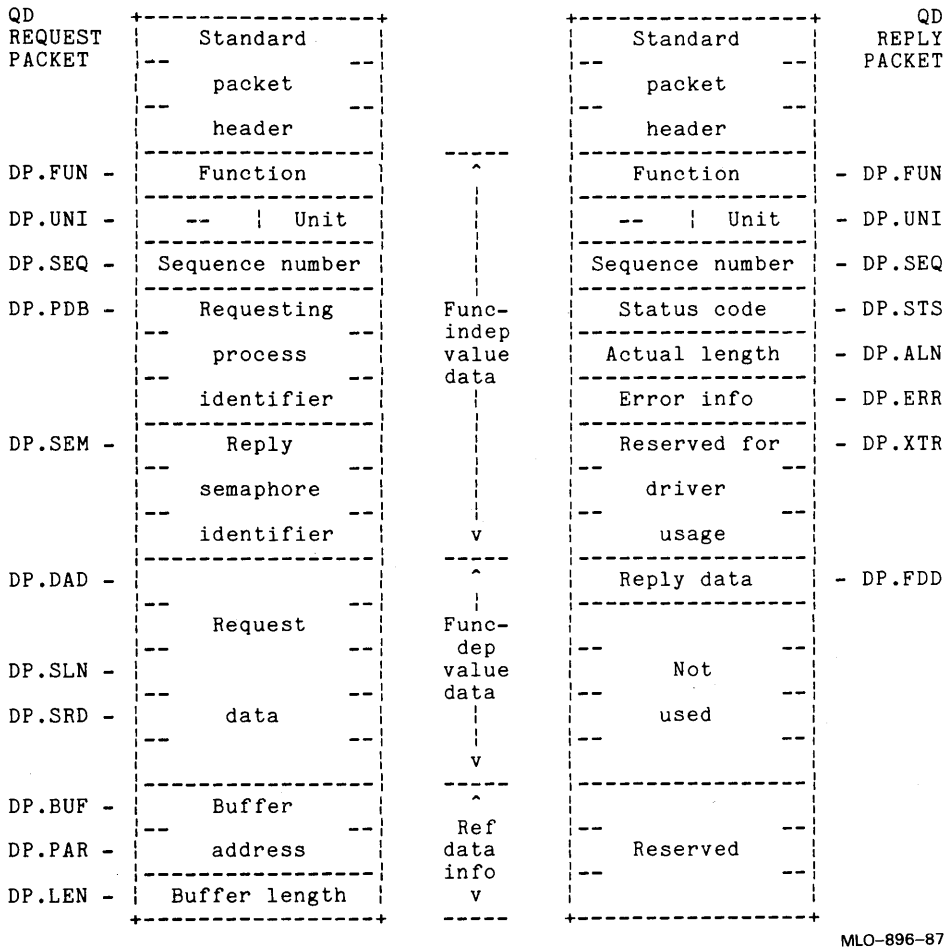
I/O requests are sent (using a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by the QD driver process.

The request queue names and number of supported units for QD driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
KXT11-CA or KXJ11-CA DMA	$QDc	1-2	0 and 1 for channels A and B

The letter c in a queue name represents a controller designation (normally A, as specified in the QD driver prefix file). The number of units configured for the controller and their unit numbers must be specified in the driver prefix file.

The general format of QD request and reply packets is shown below:



MLO-896-87

The function-independent portions of the packets shown above are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of QD driver function.

Note

The MACRO–11 field names do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

9.4.1 Read and Write Functions

The QD driver transfers data between any two of the following:

- A local address
- A Q-bus address
- An I/O device on the arbiter processor's Q-bus
- An I/O device connected directly to the KXT11–CA or KXJ11–CA (for example, the PIO chip port A via DMA channel 1)

Depending on the function-modifier bit settings, read and write operations can transfer data (FM$TSO and FM$TTS clear), search data for a user-specified value (FM$TSO set), or transfer data while searching (FM$TTS set). Read operations transfer data from the address specified in the record at DP.DAD in the request packet to the address specified in the record at DP.BUF. The address in the record at DP.DAD must contain a physical address, and the record must be equivalent to the Pascal record of type DMA$ADDRESS. This record is described in more detail below. The address in the record at DP.BUF may contain either a virtual address or a physical address. If a virtual user buffer address is specified, all addressing control bits must be defaulted (=0), and you must send the buffer address to the driver by reference, the standard technique for sending packets to drivers.

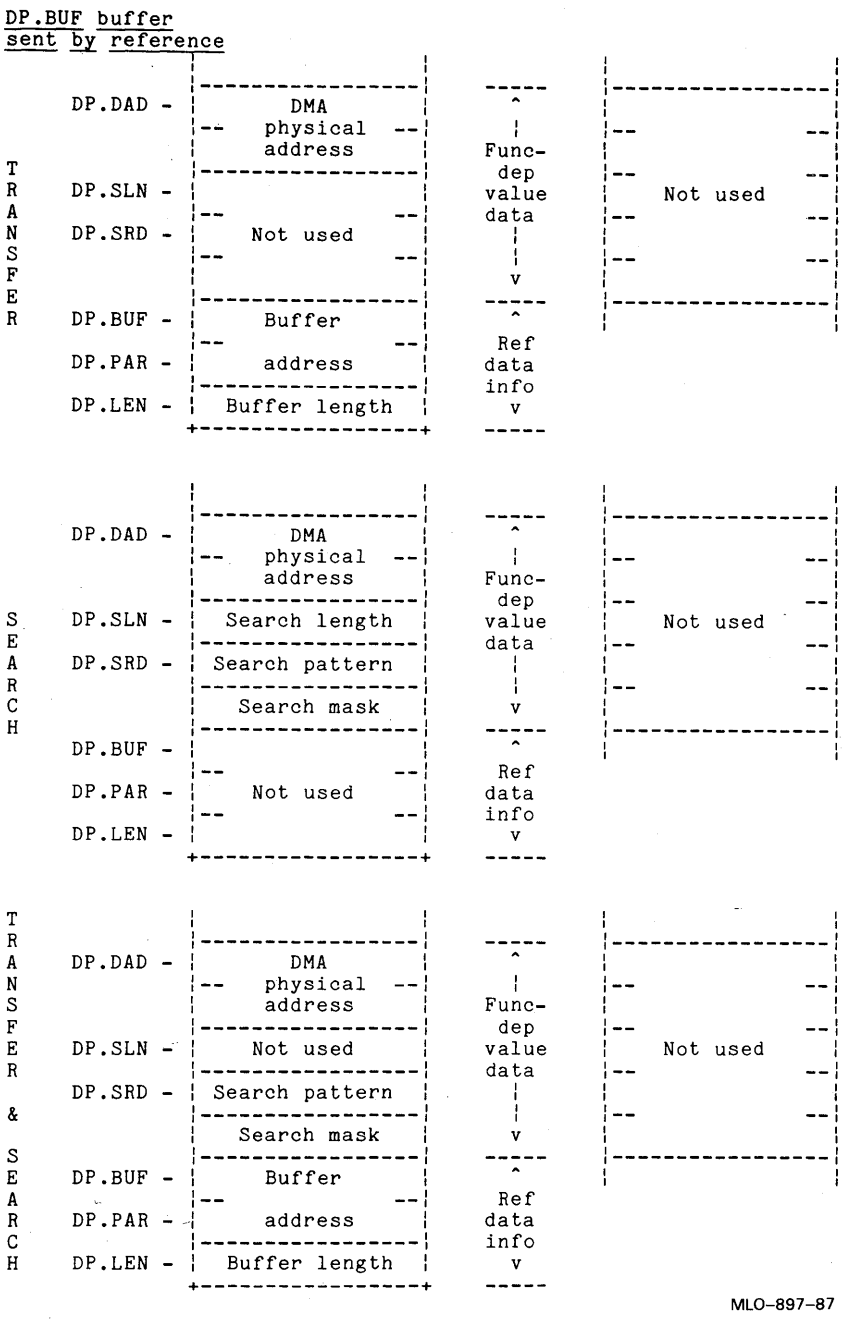
If a physical address is specified, you must fill in the address record beginning at DP.BUF and the length at DP.LEN as part of the request packet and send the entire packet by value only (with a value length six bytes larger to include the address record and the length field). The address record must be equivalent to the Pascal record of type DMA$ADDRESS. The driver checks the request packet and can tell by a bit setting in the packet header whether the buffer was sent by reference or by value. That determines whether the address record at DP.BUF contains a virtual or physical address. If the address in the record at DP.BUF is virtual, then the driver uses the virtual address and the PAR value that follows it (at DP.PAR) to convert the virtual address to a physical address. Write operations transfer data from the address specified in the record at DP.BUF in the request packet to the address specified in the record at DP.DAD. The rules about the address records are the same as they are for read operations.

Search operations use the address specified in the record at DP.DAD as the starting point for the search. The address in the record at DP.DAD must contain a physical address, and the record must be equivalent to the Pascal record of type DMA$ADDRESS. The byte count should be filled in at offset DP.SLN, immediately following the address record. For search operations, the DP.BUF field is ignored, so the request packet may be sent to the driver by value only and need not include this field.

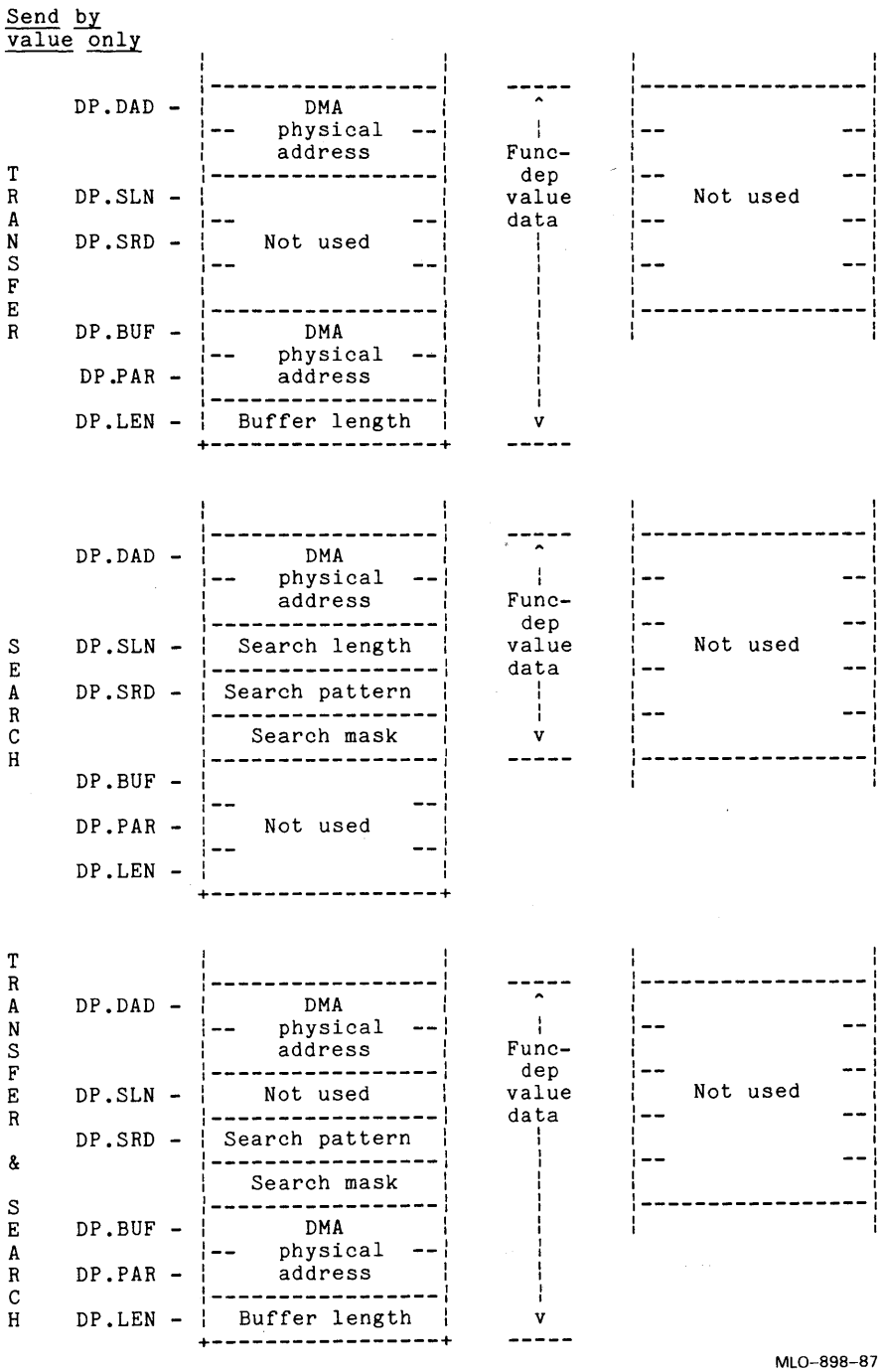
Since the packet format is rather complex, it is recommended that you use the Pascal interface routines whenever possible.

The method of accessing the address specified in a read or a write request is modified by the addressing control bits in the address record. Those addressing control bits are described below.

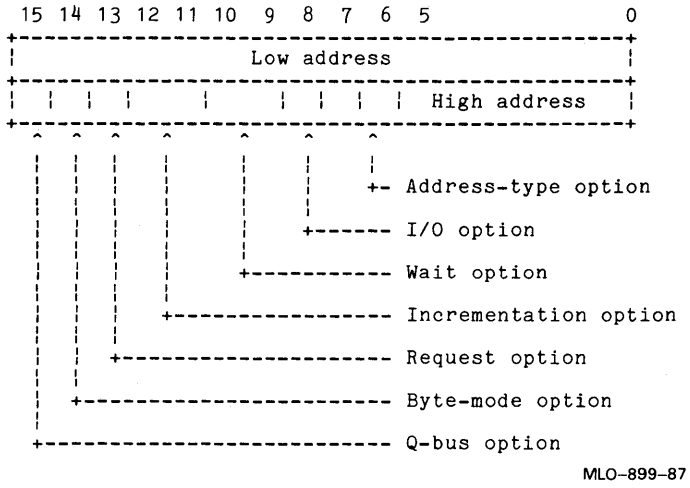
The function-dependent portions of the QD read or write request and reply packets differ for each of the three selectable operations (transfer, search, and transfer and search), as shown below:


DP.BUF buffer
sent by reference
TRANSFER
DP.DAD -
DP.SLN -
DP.SRD -
DP.BUF -
DP.PAR -
DP.LEN -
DMA
physical
address
Not used
Buffer
address
Buffer length
Func-
dep
value
data
Not used
Ref
data
info
SEARCH
Search length
Search pattern
Search mask
TRANSFER & SEARCH
MLO-897-87

When both addresses are physical



The DMA address record is a 2-word record that includes a 22-bit address and addressing control bits. The address can be a local memory address, a local or Q-bus I/O page address, or a Q-bus address. Local addresses can be virtual or physical, Q-bus addresses must be physical. The format of the DMA address record (offset DP.DAD, and possibly offset DP.BUF), which is equivalent to the Pascal record type DMA$ADDRESS, is shown below:



Note

The MACRO-11 symbols for the address-modifier field settings described below (AM$xxx) are defined by the QDDF$ macro, which resides in the COMU kernel macro library. Their Pascal equivalents, defined in QDINC.PAS, are shown in Section 9.3.1.

Proceeding from right to left in the format above:

- Address-type field, if set, means the address is physical; if clear, the address is virtual. This field is used only by the Pascal interface routines and is ignored by the QD driver. **Default:** Virtual.
- The I/O-option field, if set to AM$TIO (1), causes the DMA address to reference the I/O page rather than normal memory locations. **Default:** Not I/O page.
- The wait-option field specifies the number of wait states to be programmed into DMA-address access—zero or no wait states (AM$WS0) is the only value supported. **Default:** No wait states.
- The incrementation-option field causes the DMA address to be incremented (AM$TIA), decremented (AM$TDA), or held constant (AM$THA) during a transfer; this option can be specified independently on either address for transfer operations. **Default:** Increment address.
- The request-option field, if set to AM$WFR (1), causes the driver to wait for a hardware request (request line active) for a transfer to or from the DMA address; this is used to coordinate the DTC and the KXT11-CA/KXJ11-CA PIO port controller for DMA transfer via a PIO port (see Section 9.4.2). Since this option applies to the transfer in general and

not to just one address, it may be specified in either address record for transfer operations. **Default:** No wait for request.

- The byte-mode-option field, if set to AM$BMO (1), causes the data to be transferred in byte mode rather than word mode; byte mode is supported only for I/O to a local (internal bus) byte-oriented device—for example, a parallel port or an asynchronous line—with the other address even. (Section 9.4.2 describes DMA transfer via a parallel port.) Byte-to-word and word-to-byte funneling are not supported. Since this option applies to the transfer in general and not to just one address, it may be specified in either address record for transfer operations. **Default:** Word mode.
- The Q-bus-option field, if set to AM$TQB (1), causes the DMA address to be mapped to Q-bus space; otherwise, the internal bus is accessed. Byte mode cannot be selected when this field is set. **Default:** Not Q-bus.

Bits 15, 12 and 11, 10 and 9, 8, and 6 apply only to this address. Bits 14 and 13 apply to the operation in general and may therefore be specified in either address record for transfer operations.

The search-length (offset DP.SLN) field specifies the maximum length, in bytes, of the search to be performed for search-only operations.

If the FM$TSO function modifier is specified, bits set to 1 in the search-mask (offset DP.SRD) field mask out bits in the object word. For example, to search only the low-order byte of each word in a buffer, you should specify a search mask with all eight high-order bits set to 1. Thus, the low-order byte of each word in the buffer will be compared with the low-order byte of the search pattern (offset DP.SRD+2).

Note

To search for a byte in a buffer, you must perform two search operations. You must first search the low-order byte of each word and mask out the high, then search the high-order byte of each word and mask out the low. When the high-order byte is being searched, the search pattern must be shifted to the high-order byte.

The actual-length (offset DP.ALN) field of the read or write reply packet returns the count of bytes transferred, unless function modifier FM$TSO was specified, in which case it returns the count of bytes searched.

9.4.2 KXT11-CA/KXJ11-CA PIO DMA Process

If you want to perform DMA transfers via a KXT11-CA or KXJ11-CA parallel port, you must first set up and send a DMA Read or a DMA Write request packet to the YK driver and wait for the reply. If the reply indicates normal status, you then send a DMA transfer command to the DMA (QD) driver; otherwise, you report an error or wait. You must wait for each request to complete, since only one PIO DMA operation can be in progress at a time. After the DMA transfer completes, you send a DMA Complete request to the YK driver, which unlocks the queue of requests for that port.

For guidelines to follow when performing DMA I/O on a KXT11-CA/KXJ11-CA parallel port—and a sample Pascal program—see Section 6.4.2.4.

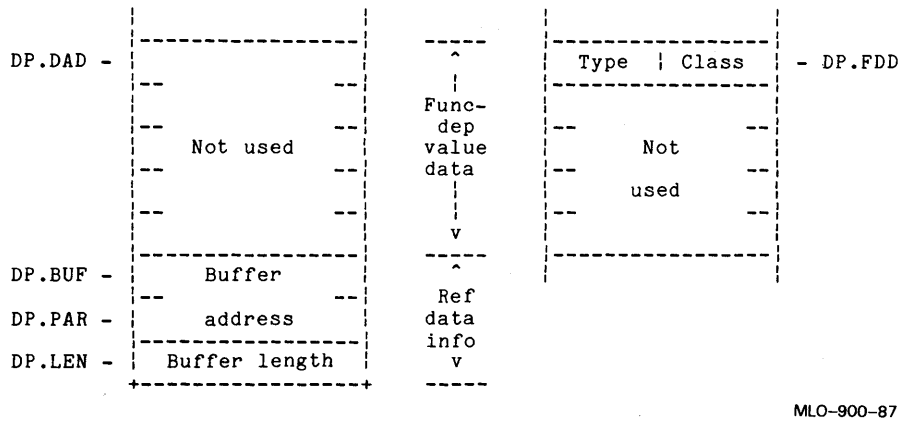
9.4.3 KXT11-CA/KXJ11-CA I/O Using SLU2A or SLU2B with DMA

You can use the DMA controller to transfer data to or from either of the serial line ports SLU2 channel A and SLU2 channel B. User-supplied software must perform the necessary setup to make this work, however, since the TT and XS drivers do not support this capability.

9.4.4 Get Characteristics Function

The Get Characteristics function returns device class and type to the reply packet and the contents of a specified unit's device registers to a user-specified buffer.

The function-dependent portions of the Get Characteristics request and reply packets are shown below:

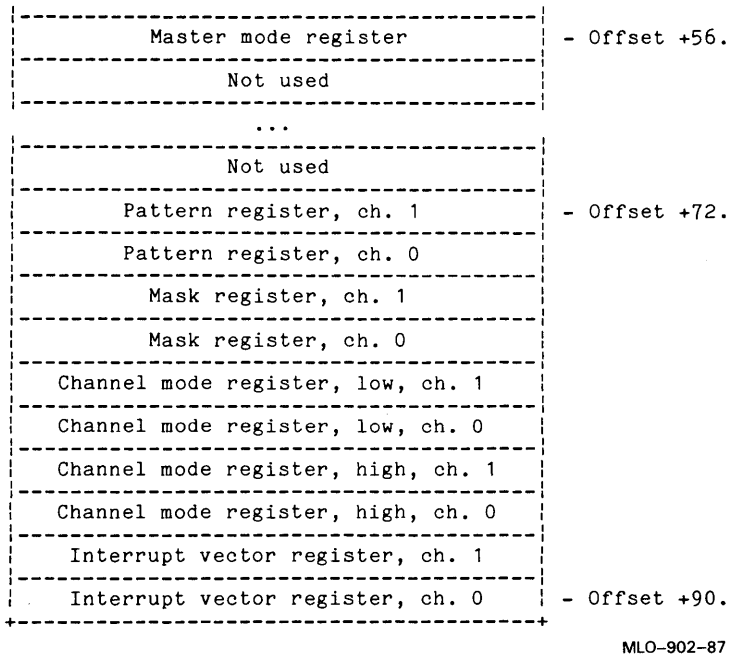


MLO-900-87

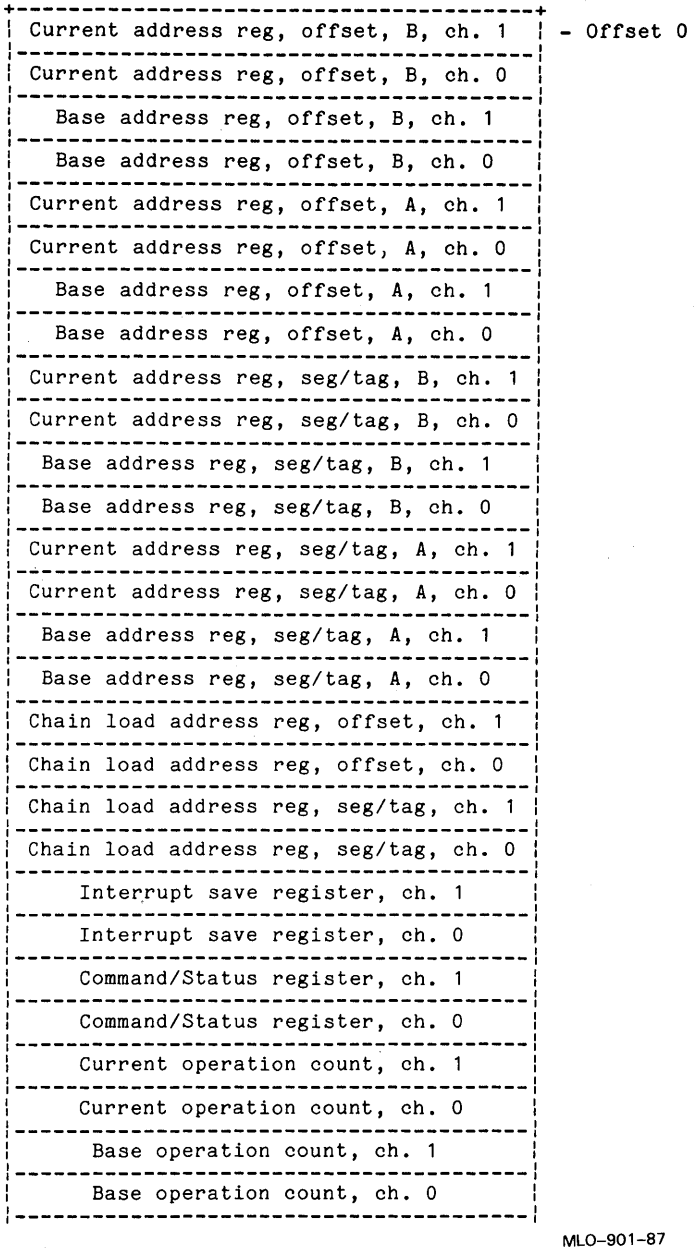
In the information above:

- Class is DC$RLT for real-time device class.
- Type is RT$QDK for KXT11-CA/KXJ11-CA DMA channel.
- Buffer address and buffer length specify the location and size of the device register buffer.

The QD driver can return up to 92 bytes—46 words—of status information. If you specify a buffer length smaller than 92, the driver returns only the number of bytes you request. The format of the status information that is returned is shown below. See the *KXT11-CA Single-Board Computer User's Guide* or *KXJ11-CA Single-Board Computer User's Guide* for detailed descriptions of the listed registers and counts.


Master mode register
- Offset +56.
Not used
...
Not used
Pattern register, ch. 1
- Offset +72.
Pattern register, ch. 0
Mask register, ch. 1
Mask register, ch. 0
Channel mode register, low, ch. 1
Channel mode register, low, ch. 0
Channel mode register, high, ch. 1
Channel mode register, high, ch. 0
Interrupt vector register, ch. 1
Interrupt vector register, ch. 0
- Offset +90.

MLO-902-87



MLO-901-87

9.4.5 Channel Allocation and Deallocation

An Allocate Channel operation dedicates a channel of the DTC for the use of a single process. The IF$ALL function directs the QD driver to reject any future requests that are issued by tasks other than the one that issued the IF$ALL request.

Deallocate Channel reverses the effect of the Allocate Channel request.

For each request, you specify the DTC channel number (offset DP.UNI). The function-dependent portions of the channel allocation and deallocation request and reply packets are not used.

9.5 Status Codes

If the QD driver detects an error during an I/O operation, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If no error is detected during the I/O operation, the driver returns a value of ES$NOR (0) in the status-code field.

The QD driver returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O abort, driver process deleted, request not serviced
ES$DAL	HARD_IO	Device (channel) already allocated
ES$IVM	HARD_IO	Invalid mode
ES$IVP	HARD_IO	Invalid parameter: odd address specified for word-mode operation, byte mode illegal for Q-bus transfer
ES$NXM	HARD_IO	Nonexistent or read-only memory
ES$NXU	HARD_IO	Nonexistent unit
ES$IFN	SOFT_IO	Invalid function code
ES$TIM	HARD_IO	(KXJ11–CA only). A SACK (Q-bus grant request) timeout occurred. This may occur if there is heavy DMA activity on the Q-bus. The actual length byte count indicates the number of bytes that were successfully transferred. Depending on your application requirements, you may be able to restart the transfer from where it left off, restart the entire transfer, or report an error.
ES$UFN	SOFT_IO	Unsupported function, file open attempted

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO–11 users.

Note

Not listed above are exception codes for kernel-detected errors that the QD driver raises rather than passing back to the requesting process.

9.6 QD Driver Prefix File

Figure 9–2 shows the QD driver prefix module. The QD driver priorities, CSR, vector, and number of units are standard for the KXT11–CA or KXJ11–CA board. The QD prefix file, as supplied, should be appropriate for all applications and not need modification.

Figure 9–2: KXT11–CA/KXJ11–CA DTC Driver Prefix File (QDPFX.MAC)

```
.nlist
        .enabl  LC
.list
.title   QDPFX - KXT11--CA/KXJ11--CA DTC Device Driver Prefix Module

;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION.
;      ALL RIGHTS RESERVED.
;
   .MCALL   DRVCF$
   .mcall   ctrcf$

QD$PPR   ==   175.       ; Process priority
QD$HPR   ==   4          ; DTC hardware priority
QD$IPR   ==   250.       ; Process initialization priority

   drvcf$   dname=QD,nctrl=1

;Specify vector and base CSR for Unit 0. Unit 0 has a vector of 224 and a
;base csr of 174402. Unit 1 has a vector of 230 and a base csr of 174400.

   ctrcf$   cname=A,nunits=2.,csrvec=<174402,224>,units=<0:1>

   .end
```

Chapter 10
Instrument Bus Driver

This chapter describes the use of the MicroPower/Pascal IEEE-488 instrument bus (XE) driver, which supports I/O operations on the IEQ11-A instrument bus interface. The IEQ11-A is a DMA controller that can interface a MicroPower/Pascal target processor to two independent instrument buses. Each bus can have up to 15 devices (including the IEQ11-A) in a sequential configuration.

10.1 Instrument Bus Features and Capabilities

The following summary of the features and capabilities of the IEEE-488 bus and the IEQ11-A interface is intended to serve as a basis for discussion of the XE driver's features and capabilities. The mnemonics listed for the bus lines and the IEQ11-A hardware registers are referenced throughout the chapter.

The IEEE-488 bus is a General Purpose Interface Bus (GPIB). ANSI/IEEE Standard 488-1978, "IEEE Standard Digital Interface for Programmable Instrumentation," specifies the characteristics of the bus and the functions it must perform.

The bus consists of 24 lines. Of these, eight lines are ground wires, and 16 carry information. Of the 16 information lines, three are used for handshaking control, and five for bus management; eight carry data between devices on the bus.

You will generally not be concerned with the control lines (NRFD, DAV, and NDAC), since the hardware takes care of the handshaking.

The five bus management lines are:

Line	Mnemonic
Attention	ATN
Service request	SRQ
Interface clear	IFC
End or identify	EOI
Remote enable	REN

The eight data lines are used to transfer a byte of data at a time across the bus.

At any time, one and only one device on the bus will act as bus controller. The bus controller issues the commands needed to perform all data transfers. Each device on the bus has the potential to perform the following functions:

- Act as bus controller
- Act as "talker" in a bus transfer
- Act as "listener" in a bus transfer
- Issue a service request to the bus controller
- Respond to polls by the bus controller

The IEQ11-A provides two independent ports to the IEEE-488 bus. These ports can interface to two different buses or provide two ports into the same bus. Note that the ports are treated as separate controllers—not as two units of an IEQ11-A controller—for the purposes of configuring MicroPower/Pascal applications.

The functioning of these ports is controlled by the use of eight hardware registers for each port. The registers are:

Register	Mnemonic	Address
IEEE Status Read: Address Status/Bus Status Write: Int Mask 0/Int Mask 1	ISR	76XXX0
IEEE Interrupt Read: Int Status 0/Int Status 1 Write: -/Address	IIR	76XXX2
IEEE Command Read: Cmd Pass Thru/- Write: Serial Poll/Auxiliary Cmd	ICR	76XXX4
IEEE Data Read: -/Data In Write: Parallel Poll/Data Out	IDR	76XXX6
Control/Status	CSR	76XX10
Bus Address	BAR	76XX12
Byte Count	BCR	76XX14
Match Character	MCR	76XX16

The corresponding registers for the two ports have identical addresses. The setting of a multiplexer bit in the CSR—based on a user-specified controller ID or unit number—determines which port's register is referenced. Aside from register sharing, however, the two instrument bus ports are functionally independent.

As indicated by the Read and Write designations above, the four "IEEE" register addresses reference different registers, depending on whether a reference is a read or a write.

10.2 Driver Features and Capabilities

The instrument bus (XE) driver provides the basic mechanisms for control of the instrument bus interface. Among the operations supported are DMA data transfers, bus controller operations, service requests, serial polling, and parallel polling.

All IEQ11-A data transfers are DMA. Before a transfer, the device that will send the data is addressed as "talker," and any device to receive the data is addressed as "listener." (A read or write consists of enabling the DMA transfer and dropping the ATN line.)

10.3 Performing Instrument Bus I/O

For most MicroPower/Pascal applications, you perform instrument bus I/O by invoking Pascal support routines—WRITE_IEQ, IEQ_PASS_CONTROL, IEQ_REQ_SERVICE, IEQ_SERIAL, and so forth. Those routines provide high-level nonfile access to the IEQ11–A instrument bus ports. The XE support routines issue Pascal send requests to the request queue semaphore of the XE driver. The routines are described in Section 10.4.

Note

You cannot perform file-oriented Pascal operations on the instrument bus ports. If you try to open an instrument bus file, the XE driver returns an unsupported function (ES$UFN) exception code and the OTS raises the exception (unless you requested a status return).

In addition to invoking the Pascal support routines, you must:

1. Edit the DEVICES macro in the system configuration file to reflect the instrument bus port interrupt vector addresses
2. Edit the XE driver prefix file to reflect:
   - Number of IEQ11–A controllers (instrument bus ports)—up to two per IEQ11–A
   - [For each port:] Controller identifier (A, B, ...), CSR address, interrupt vector address, and number of controller units (1)
   - Hardware interrupt priority
   - Driver initialization and request-handling process priorities
3. Build into your application the following I/O system components:
   - XE driver process
   - Pascal instrument bus support routines (from kit files XESUB.PAS and XEINC.PAS)

For more information on setting up your application software for instrument bus I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 10.8 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

As an alternative to using the Pascal support routines for instrument bus I/O, you can issue your own Pascal or MACRO–11 packet-level requests to the driver (low-level nonfile access).

The following sections describe the Pascal support routine interface to the XE driver, the lower-level request/reply packet interface, the status codes and extended error information that can be returned to users of either interface, and the XE driver prefix file.

10.4 Pascal Support Routine Interface

The following support routines, written in Pascal and independent of the file system, provide a high-level interface to the IEQ11-A instrument bus ports:

- READ_IEQ procedure
- WRITE_IEQ procedure
- SET_STATE function
- WRITE_EOI_IEQ procedure
- IEQ_COMMAND procedure
- IEQ_SERIAL procedure
- IEQ_PARALLEL_POLL procedure
- IEQ_PARALLEL_LOAD procedure
- IEQ_PARALLEL_CONFIG procedure
- IEQ_AUX_COMMAND procedure
- IEQ_REQ_SERVICE procedure
- IEQ_CONTROL_GTS procedure
- IEQ_PASS_CONTROL procedure
- SET_INT_MASK procedure
- REC_IEQ_EVENT procedure

The XE support routines allow you to use all the packet-level driver functions except the following:

- Get Characteristics (IF$GET)
- Serial Poll Over All Devices (IF$SPO)
- Get Control (IF$GTC)
- Recognize Event (IF$RES)

To perform the functions not performed by support routines, either use the lower-level request/reply interface directly or write Pascal procedures that take a user-specified queue semaphore ID and send the appropriate request packet to the driver. For the Get Characteristics (IF$GET) function, you can use the Get Characteristics function (descriptor version) in the distribution kit file GETSET.PAS. That function issues a Get Characteristics (IF$GET) request packet to the driver.

The following sections describe the Pascal routines for instrument bus I/O. Each routine allocates an I/O packet, fills it in with information based on the function parameters, sends it to the XE driver queue semaphore for the specified port, and returns immediately to the caller. If the routine has a reply parameter, the driver sends a standard driver reply via the specified queue semaphore when the operation is complete. (The driver reply packets are described in Section 10.5.)

The following files on the MicroPower/Pascal distribution kit are required for using the routines:

File	Description
XESUB.PAS	Instrument bus routine source module
XEINC.PAS	Instrument bus routine include file
IOPKTS.PAS	Pascal I/O include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

10.4.1 READ_IEQ

The READ_IEQ procedure requests a read operation. The caller receives data in DMA mode from the device addressed as "talker."

In order for a device to be able to receive data, it must be addressed as "listener." If the unit is controller-in-charge but not a listener when a read request is issued, the driver addresses the unit as listener. If addressed in this manner, the unit will automatically be deaddressed after the request has completed.

There are three ways in which a read request may terminate:

- Byte count overflow
- EOI termination
- Match character termination

Match character detection allows you to stop a transfer upon detection of a given number of consecutive end-of-string (EOS) characters. You specify the EOS character, which depends on the device that is currently the talker.

The packet-level equivalent of READ_IEQ is the IF$RDL function.

The syntax for calling the procedure is as follows:

```
READ_IEQ ( unit, unit_desc, buffer, leng, mlength, chr, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR buffer	PACKED ARRAY [first..last: INTEGER] of CHAR	Data buffer
leng	INTEGER	Buffer length
mlength	INTEGER	Match length up to 63 (decimal); 0 disables match character processing
chr	CHAR	Match character
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The count of bytes transferred is returned in the actual-length field of the XE driver reply packet.

10.4.2 WRITE_IEQ

The WRITE_IEQ procedure requests a write operation without EOI termination. A write operation transmits data in DMA mode to all devices addressed as "listener."

In order for a device to be able to send data, it must be addressed as "talker." If the unit is not addressed as talker when a write request is issued, the driver automatically addresses the unit as talker, using the auxiliary command Talk Only (TON) and deaddresses the unit after the request has completed.

The packet-level equivalent of WRITE_IEQ is the IF$WTL function.

The syntax for calling the procedure is as follows:

```
WRITE_IEQ ( unit, unit_desc, buffer, leng, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR buffer	PACKED ARRAY [first..last: INTEGER] of CHAR	Data buffer
leng	INTEGER	Buffer length
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The count of bytes transferred is returned in the actual-length field of the XE driver reply packet.

10.4.3 SET_STATE

The SET_STATE function can be used to perform one or more of the following state alterations, listed in the order they are performed:

- Issue master clear
- Set system controller bit in CSR
- Clear system controller bit in CSR
- Set primary address
- Send software reset
- Set controller to controller active state

Master clear resets the IEQ11–A controller without affecting other devices in the system.

Setting the system controller bit makes the specified controller the system controller.

Setting the primary address loads a user-specified address into the IEEE command register (ICR).

Software reset resets all of the TMS 9914 registers.

SET_STATE always returns the BOOLEAN value TRUE.

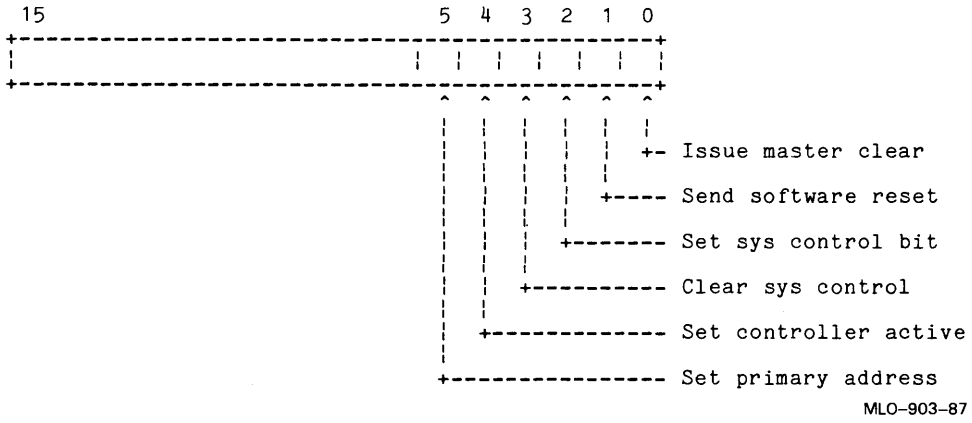
The packet-level equivalent of SET_STATE is the IF$SET function.

The syntax for calling the function is as follows:

```
SET_STATE ( comm, unit, unit_desc, address )
```

Parameter	Type	Description
comm	INTEGER	Command mask
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
address	INTEGER	Primary address, supplied only for the set primary address operation

The mask word has the following format:



10.4.4 WRITE_EOI_IEQ

The WRITE_EOI_IEQ procedure requests a write operation with End or Identify (EOI) termination. A write operation transmits data in DMA mode to all devices addressed as "listener." The EOI line is reset after the last byte is transmitted.

In order for a device to be able to send data, it must be addressed as "talker." If the unit is not addressed as talker when a write request is issued, the driver automatically addresses the unit as talker, using the auxiliary command Talk Only (TON) and deaddresses the unit after the request has completed.

The packet-level equivalent of WRITE_EOI_IEQ is the IF$WLE function.

The syntax for calling the procedure is as follows:

```
WRITE_EOI_IEQ ( unit, unit_desc, buffer, leng, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR buffer	PACKED ARRAY [first..last: INTEGER] of CHAR	Data buffer
leng	INTEGER	Buffer length
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The count of bytes transferred is returned in the actual-length field of the XE driver reply packet.

10.4.5 IEQ_COMMAND

The IEQ_COMMAND procedure requests a Write IEEE Remote Messages operation, which outputs a buffer of command bytes via the data output register. The commands are grouped as follows:

Values	Command Group
0–17	Address
20–37	Universal
40–77	Listen Address
100–137	Talk Address
140–177	Secondary

The *IEQ11–A User's Guide* lists and describes the commands in each group.

A Write IEEE Remote Messages request can be issued only by the controller-in-charge. In order to output data via the data output register, the unit must be in controller active state (CACS). If the unit is not in CACS when the request is issued, the driver attempts to enter that state, issuing an error if it fails.

The packet-level equivalent of IEQ_COMMAND is the IF$CMD function.

The syntax for calling the procedure is as follows:

```
IEQ_COMMAND ( unit, unit_desc, comm, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR comm	ARRAY[first..last: INTEGER] of BYTE_RANGE	Buffer of command bytes
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

10.4.6 IEQ_SERIAL

The IEQ_SERIAL procedure requests that a serial poll be performed while the service request (SRQ) bit is set. A serial poll is used by the controller-in-charge to determine which devices have issued a service request.

A serial poll operation begins with the deactivation of all listeners. Each device specified in the user's buffer is then addressed as "talker" for the purpose of reading its status byte. If bit 7 of a status byte is set, the currently addressed talker was requesting service. The status byte is copied to the user's buffer. If the SRQ bit is still set, the next device is addressed and the poll continues; otherwise, the poll terminates.

The packet-level equivalent of IEQ_SERIAL is the IF$SPS function.

The following data types from XEINC.PAS are referenced below:

```
TYPE
  ieq_address = PACKED RECORD
        prim_addr,
        sec_addr : [BYTE] BYTE_RANGE;
      END;

  ser_poll_buf = PACKED RECORD
        addr : ieq_address;
        status : INTEGER;
      END;
```

The syntax for calling the procedure is as follows:

```
IEQ_SERIAL ( unit, unit_desc, len, buff, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
len	INTEGER	Buffer length (currently ignored by driver, which performs its own buffer length calculation)
VAR buff	ARRAY[first..last: INTEGER] of ser_poll_buf	Buffer of device/status parameter pairs
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The first word of each parameter pair in the buffer gives a talker address. Extended addressing is not used; however, you can write your own variation of the procedure to use extended addressing. (See the description of the IF$SPS function in Section 10.5.6.)

The talker's status byte is returned to the second word of each pair.

10.4.7 IEQ_PARALLEL_POLL

The IEQ_PARALLEL_POLL procedure requests a parallel poll operation, used by the controller-in-charge to poll up to eight devices on the IEEE bus. The driver loops for 100 microseconds to wait for the parallel poll to complete. The result of a parallel poll is a status byte that is returned in an XE driver reply message—specifically, the low-order byte of the first word of the function-dependent reply data. The byte contains a bit for each of eight devices selected via the IEQ_PARALLEL_CONFIG procedure.

The packet-level equivalent of IEQ_PARALLEL_POLL is the IF$PPO function.

The syntax for calling the procedure is as follows:

```
IEQ_PARALLEL_POLL ( unit, unit_desc, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

10.4.8 IEQ_PARALLEL_LOAD

The IEQ_PARALLEL_LOAD procedure requests that the parallel poll register be loaded with a status byte. When a parallel poll is conducted, this information is transferred to the controller-in-charge. You should update this byte to reflect the state of the device.

The packet-level equivalent of IEQ_PARALLEL_LOAD is the IF$LPP function.

The syntax for calling the procedure is as follows:

```
IEQ_PARALLEL_LOAD ( unit, unit_desc, stat );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
stat	INTEGER	Status byte to be placed in parallel poll register

The format of the status byte is determined by an IEQ_PARALLEL_CONFIG operation.

10.4.9 IEQ_PARALLEL_CONFIG

The IEQ_PARALLEL_CONFIG procedure requests a Parallel Poll Configure operation, used by the controller-in-charge to configure other devices on the IEEE bus for a parallel poll. It instructs those devices on how to respond to a parallel poll.

The packet-level equivalent of IEQ_PARALLEL_CONFIG is the IF$PPC function.

The following data type from XEINC.PAS is referenced below:

```
TYPE
  pp_buffer = packed record
                l_addrs : [byte] byte_range;
                cfg_byte : [byte] byte_range;
              end;
```

The syntax for calling the procedure is as follows:

```
IEQ_PARALLEL_CONFIG ( unit, unit_desc, buff, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
buff	ARRAY[first..last: INTEGER] of pp_buffer	Buffer of up to eight listener address/ configuration-byte parameter pairs
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

Each parameter pair in the buffer gives a listener address in the range 40 to 76 (octal) and a configuration byte with a value in the range 0 to 7. The listener address selects a device and the configuration byte value determines which bit in the parallel poll register the device will update to reflect its state. The state of that bit is returned to the controller-in-charge when a parallel poll occurs.

10.4.10 IEQ_AUX_COMMAND

The IEQ_AUX_COMMAND procedure requests that an auxiliary command be issued by writing a byte to the auxiliary command register. Auxiliary commands are used to enable and disable most of the selectable features of IEQ11–A registers.

The packet-level equivalent of IEQ_AUX_COMMAND is the IF$AUX function.

The syntax for calling the procedure is as follows:

```
IEQ_AUX_COMMAND ( unit, unit_desc, aux_comm, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
aux_comm	INTEGER	Auxiliary command
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The auxiliary commands are described in detail in the *IEQ11-A User's Guide*. Their values are shown below:

Value	Command
0.	Software reset
1.	Release ACDS holdoff
2.	Release RFD holdoff
3.	Holdoff all data
5.	Set new byte available false (bit 7 not applicable)
8.	Assert force end or identify (FEOI) on next byte
9.	Listen only
10.	Talk only
11.	Go to standby
12.	Take control asynchronously
13.	Take control synchronously
14.	Request parallel poll
15.	Send interface clear
16.	Send remote enable
17.	Request control
18.	Release control
20.	Pass through next secondary
24.	Service request bit 2
Bit 7	Clear/set bit (required for clear/set type of features)

10.4.11 IEQ_REQ_SERVICE

The IEQ_REQ_SERVICE procedure performs a Request Service operation, which generates a service request to the controller-in-charge and places a user-provided status byte in the serial poll register. The controller-in-charge detects that the SRQ line has asserted and performs a serial poll to determine which device requested service. When the requesting device is addressed in a serial poll, the controller-in-charge reads the status byte from the serial poll register. The meaning of the status byte is application dependent, but it should indicate to the controller-in-charge the type of servicing required.

Note

Bit 6 of the status byte is used to set the SRQ line and is not available to the user's program.

The packet-level equivalent of IEQ_REQ_SERVICE is the IF$RSV function.

The syntax for calling the procedure is as follows:

```
IEQ_REQ_SERVICE ( unit, unit_desc, stat );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
stat	INTEGER	Status byte

10.4.12 IEQ_CONTROL_GTS

The IEQ_CONTROL_GTS procedure requests a Go to Standby operation, used by the controller-in-charge to go from controller active state (CACS) to controller standby state (CSBS). If the unit is already in CSBS when the request is issued, no action is taken. A state error is generated if the unit is not controller-in-charge.

Since the driver automatically goes to the state required to process a request, this procedure is not normally needed.

The packet-level equivalent of IEQ_CONTROL_GTS is the IF$GTS function.

The syntax for calling the procedure is as follows:

```
IEQ_CONTROL_GTS ( unit, unit_desc, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

10.4.13 IEQ_PASS_CONTROL

The IEQ_PASS_CONTROL procedure passes control to a user-specified device, which takes control in controller active state. The device is addressed to receive control as "talker," a handshake is completed, and control is passed (by releasing the ATN line).

The packet-level equivalent of IEQ_PASS_CONTROL is the IF$PCT function.

The syntax for calling the procedure is as follows:

```
IEQ_PASS_CONTROL ( unit, unit_desc, t_addr, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
t_addr	INTEGER	Talker address in the range 100 to 136 (octal)
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

10.4.14 SET_INT_MASK

The SET_INT_MASK procedure sets up event recognition by specifying events for which recognition is requested. Those events are subsequently detected and reported to the user. (See the REC_IEQ_EVENT procedure.) The events are specified in a bit mask. If a bit is set in the mask, recognition of the corresponding event is requested.

Normally, the controller-in-charge requests recognition of a service request, and devices that are not controller-in-charge request recognition of all the other events.

Event recognition remains enabled until the procedure is called with an argument of 0.

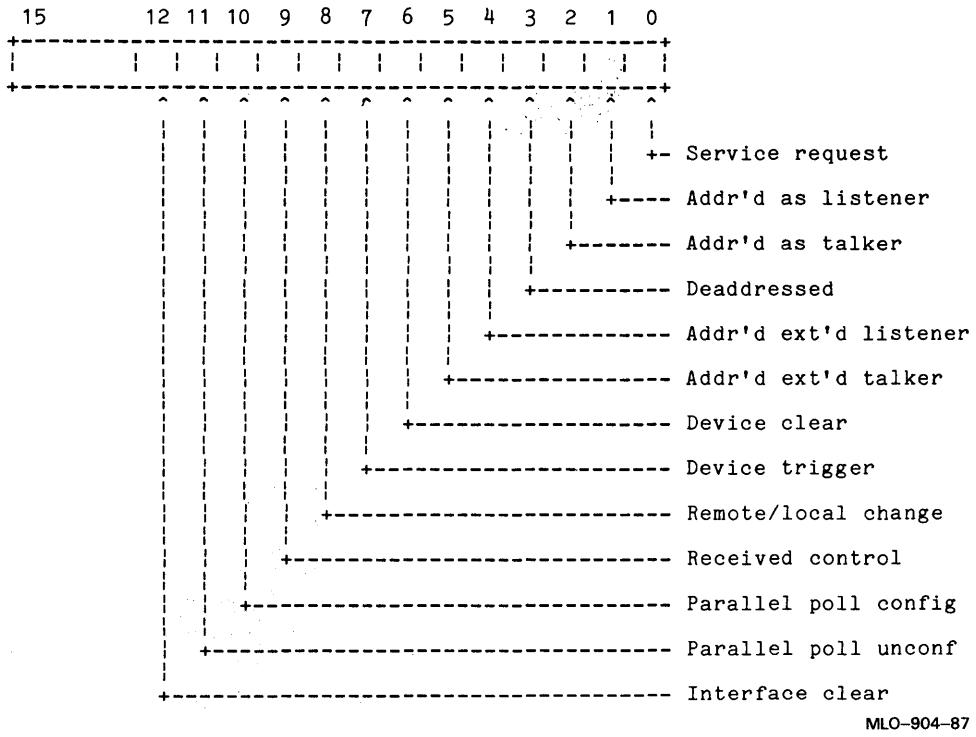
The packet-level equivalent of SET_INT_MASK is the IF$SEM function.

The syntax for calling the procedure is as follows:

```
SET_INT_MASK ( unit, unit_desc, mask );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
mask	INTEGER	Event mask

The event mask has the following format:



If recognition of an event represented by bits 1–9 is requested and the event occurs, the IEEE–488 is locked until released by the program. The bus can be released only by an IEQ_AUX_COMMAND call that releases ACDS holdoff (aux_comm = 1) or when a new request is issued to the controller, in which case the driver automatically releases the bus.

10.4.15 REC_IEQ_EVENT

The REC_IEQ_EVENT procedure returns notification of an event to the user. Event recognition must previously have been enabled with a SET_INT_MASK call. If the event has already occurred, the request is returned immediately with notification. Otherwise, the request waits until the event occurs.

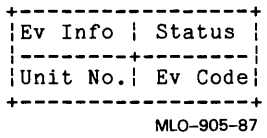
The packet-level equivalent of REC_IEQ_EVENT is the IF$REV function.

The syntax for calling the procedure is as follows:

```
REC_IEQ_EVENT ( unit, unit_desc, reply );
```

Parameter	Type	Description
unit	INTEGER	Unit number (valid range is 0 through n–1 for n configured IEQ11–A ports)
VAR unit_desc	STRUCTURE_DESC	Initialized driver queue semaphore descriptor
VAR reply	STRUCTURE_DESC	Optional initialized reply queue semaphore descriptor; if specified, it is the user's responsibility to wait for the reply

The event notification takes the form of a status block that is returned in the XE driver reply packet (first two words of the function-dependent portion—see Section 10.5.16). The return status block has the following format:



MLO-905-87

The status code field has the value ES$NOR (0).

The event code indicates the event that has occurred as follows:

Value	Event
1.	Service request
2.	Addressed as listener
3.	Addressed as talker
4.	Deaddressed
5.	Addressed as extended listener
6.	Addressed as extended talker
7.	Device clear
8.	Device trigger
9.	Remote/local change
10.	Received control
11.	Parallel poll configure
12.	Parallel poll unconfigure
13.	Interface clear

The event information byte contains the following event-dependent data:

- If extended listener or talker, the extended address

- If a remote/local change occurred, 0 if the unit is in local mode, or 1 if the unit is in remote mode
- If a Parallel Poll Configure occurred, the PPE or PPD byte
- For all other events, 0

10.5 Request/Reply Packet Interface

The packet-level functions provided by the XE driver are listed below by symbolic and decimal function code:

Code	Function
IF$RDL (1)	Read Logical
IF$WTL (4)	Write Logical
IF$SET (6)	Set Characteristics (Set State)
IF$GET (7)	Get Characteristics (Sense State)
IF$WLE (24)	Write with EOI Termination
IF$CMD (25)	Write IEEE Remote Messages
IF$SPS (26)	Serial Poll While SRQ is Set
IF$SPO (27)	Serial Poll Over All Devices
IF$PPO (28)	Parallel Poll
IF$LPP (29)	Load Parallel Poll Register
IF$PPC (30)	Parallel Poll Configure
IF$AUX (31)	Auxiliary Command
IF$RSV (32)	Request Service
IF$GTC (34)	Get Control
IF$GTS (35)	Go to Standby
IF$PCT (36)	Pass Control
IF$SEM (37)	Set Event Mask
IF$REV (38)	Wait for Event
IF$RES (39)	Recognize Event

If a request is received for an Open (IF$LOK or IF$ENT), the driver returns an unsupported function status code (ES$UFN). This will cause the Pascal OTS to raise an exception, provided that the OTS/ACP issued the Open request and the user's OPEN statement did not specify a status return.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the include files IOPKTS.PAS and XEINC.PAS.

The function modifiers recognized by the XE driver are shown below by symbolic code and bit position:

Code	Function
FM$LIS (bit 12)	Leave in state (do not reset previous state) after data transfer
FM$SRQ (bit 13)	Terminate data transfer on service request (SRQ)
FM$EAD (bit 13)	Use extended addressing in serial poll
FM$TCS (bit 14)	Take control synchronously (go to controller active state)
FM$BSM (bit 13)	Signal binary/counting semaphore

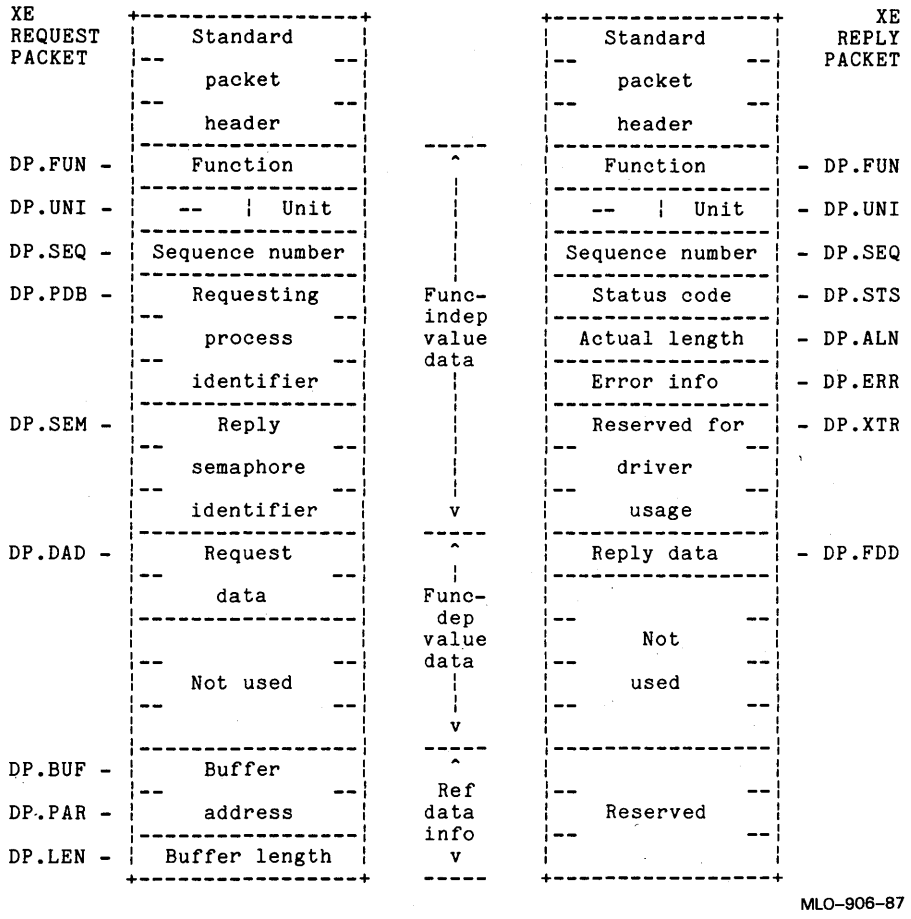
The XE driver consists of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each controller configured. I/O requests intended for a particular controller (IEQ11-A port) are sent (using a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

The request queue name and number of supported units for XE driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
Instrument bus	$XEc	1(per controller)	Sequentially upward from 0 in prefix file order, crossing controller and board boundaries

The letter c in a queue name represents a controller designation (A, B, ..., as specified in an XE driver prefix file).

The general format of the XE request and reply packets is shown below:



The function-independent portions of the packets shown above are described in the request/reply packet interface section of Chapter 1. The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The following sections describe the function-dependent portions of the request and reply packets for each type of XE driver function.

Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

10.5.1 Read Logical Function

The Read Logical (IF$RDL) function receives data from the device addressed as "talker." Note that the IEQ11-A interface is capable of receiving data in either processor or DMA mode. However, the XE driver uses only the DMA mode. Status information may be exchanged between devices by using polling requests, but the only method of receiving data is via reads.

In order for a device to be able to receive data, it must be addressed as "listener." If the unit is controller-in-charge but not a listener when a read request is issued, the driver addresses the unit as listener. If addressed in this manner, the unit will automatically be deaddressed after the request has completed.

There are three ways in which a read request may terminate:

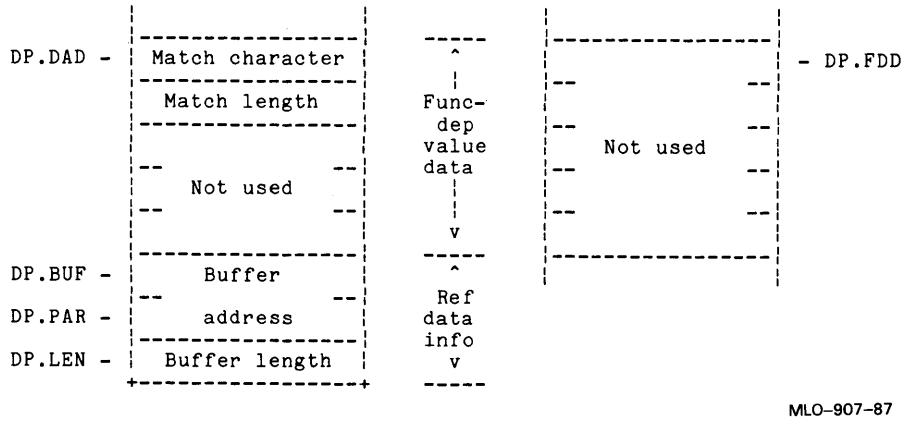
- Byte count overflow
- EOI termination
- Match character termination

Match character detection allows you to stop a transfer upon detection of a given number of consecutive end-of-string (EOS) characters. You specify the EOS character, which depends on the device that is currently the talker.

A data transfer terminates and an error is returned if an Incomplete Handshake Error (ERR) or Interface Clear Received (IFC) interrupt occurs.

If the function modifier FM$SRQ is set, a Service Request (SRQ) interrupt will terminate the data transfer. The status code ES$STD and the extended error code IE.SRQ are returned in the reply message.

The function-dependent portions of the read request and reply packets are shown below:



MLO-907-87

The match-character (low-order byte) and match-length fields specify the match character and the match-character count (up to 63 decimal) for match-character processing. A match length of 0 disables the feature.

10.5.2 Write and Write with EOI Termination Functions

The write functions (IF$WTL and IF$WTE) transmit data to all devices addressed as "listener." Note that the IEQ11-A interface is capable of transferring data in either processor request or DMA mode. However, the XE driver uses only the DMA mode. Status information may be exchanged between devices by using the polling requests, but the only method of transmitting data is via writes.

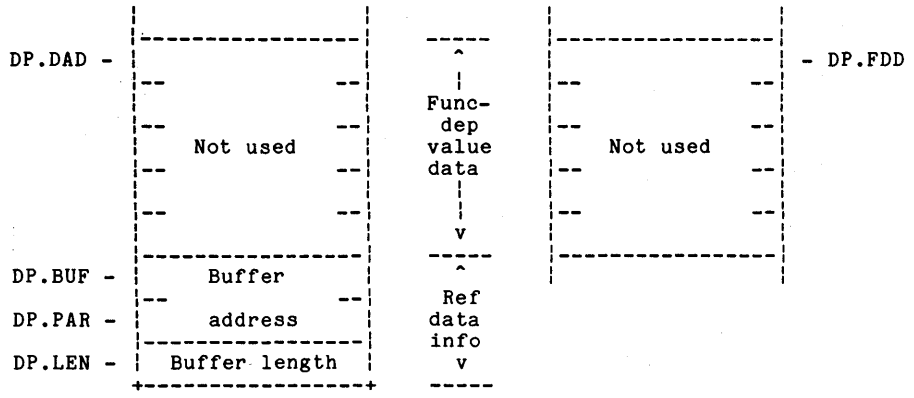
The IF$WLE request is identical to the IF$WTL request, except that it issues the auxiliary command Force End or Identify (FEOI) before transmitting the last data byte. That causes the EOI message to be sent with the last data byte. The EOI line is then reset.

In order for a device to be able to send data, it must be addressed as "talker." If the unit is not addressed as talker when a write request is issued, the driver automatically addresses the unit as talker, using the auxiliary command Talk Only (TON) and deaddresses the unit after the request has completed.

A data transfer terminates and an error is returned if an Incomplete Handshake Error (ERR) or Interface Clear Received (IFC) interrupt occurs.

If the function modifier FM$SRQ is set, a Service Request (SRQ) interrupt will terminate the data transfer. The status code ES$STD and the extended error code IE.SRQ are returned in the reply message.

The function-dependent portions of the write request and reply packets are shown below:

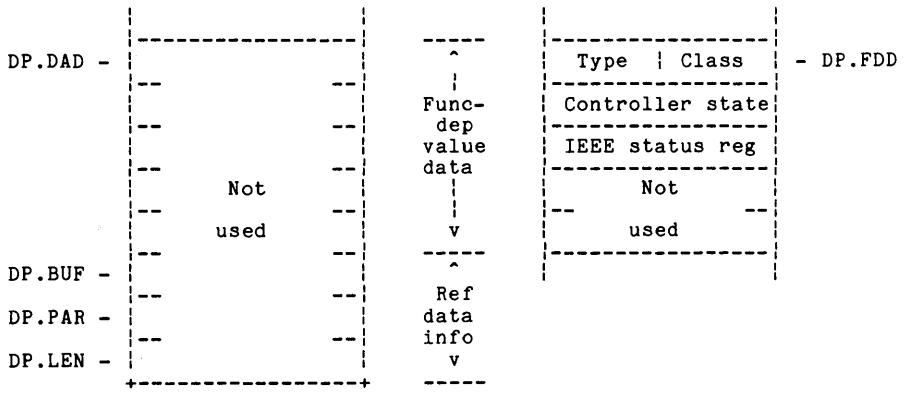


MLO-908-87

10.5.3 Get Characteristics (Sense State) Function

The Get Characteristics (IF$GET) function returns the controller state and IEEE-status-register contents in two status words.

The function-dependent portions of the Get Characteristics request and reply packets are shown below:



The possible controller-state values in the high-order byte of the first status word are shown below:

Value	State
1	Controller idle
2	Controller active
3	Controller standby

The contents of the IEEE status register (read-only) are diagrammed and described in the *IEQ11–A User's Guide.*

10.5.4 Set Characteristics (Set State) Function

The Set Characteristics (IF$SET) function can be used to perform one or more of the following state alterations, listed in the order they are performed:

- Issue master clear
- Set system controller bit in CSR
- Clear system controller bit in CSR
- Set primary address
- Send software reset
- Set controller to controller-active state

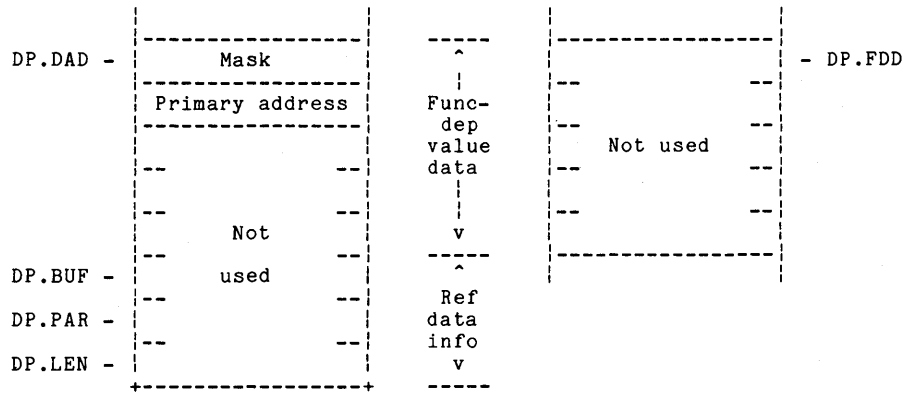
Master clear resets the IEQ11–A controller without affecting other devices in the system.

Setting the system controller bit makes the specified controller the system controller.

Setting the primary address loads a user-specified address into the IEEE command register (ICR).

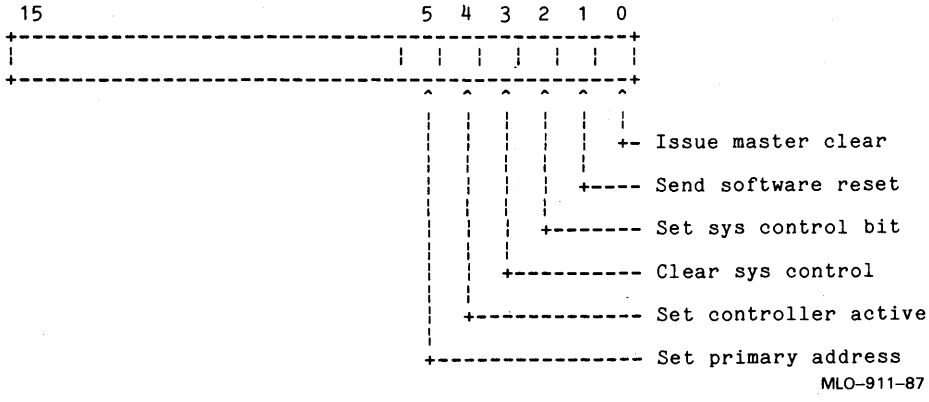
Software reset resets all of the TMS 9914 registers.

The function-dependent portions of the Set Characteristics request and reply packets are shown below:



The primary address is supplied for the set primary address operation only.

The mask word has the following format:



10.5.5 Write IEEE Remote Messages Function

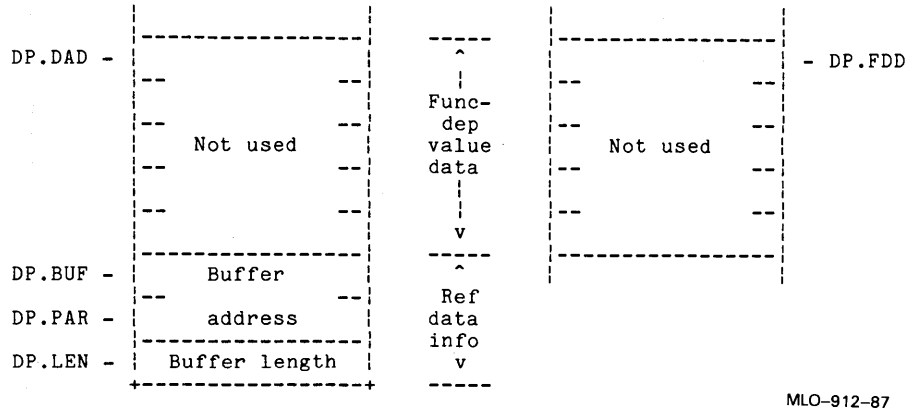
The Write IEEE Remote Messages (IF$CMD) function outputs a buffer of command bytes via the data output register. The commands are grouped as follows:

Values	Command Group
0–17	Address
20–37	Universal
40–77	Listen Address
100–137	Talk Address
140–177	Secondary

The *IEQ11-A User's Guide* lists and describes the commands in each group.

The IF$CMD request can be issued only by the controller-in-charge. In order to output data via the data output register, the unit must be in controller active state (CACS). If the unit is not in CACS when the request is issued, the driver attempts to enter that state, issuing an error if it fails.

The function-dependent portions of the IF$CMD request and reply packets are shown below:



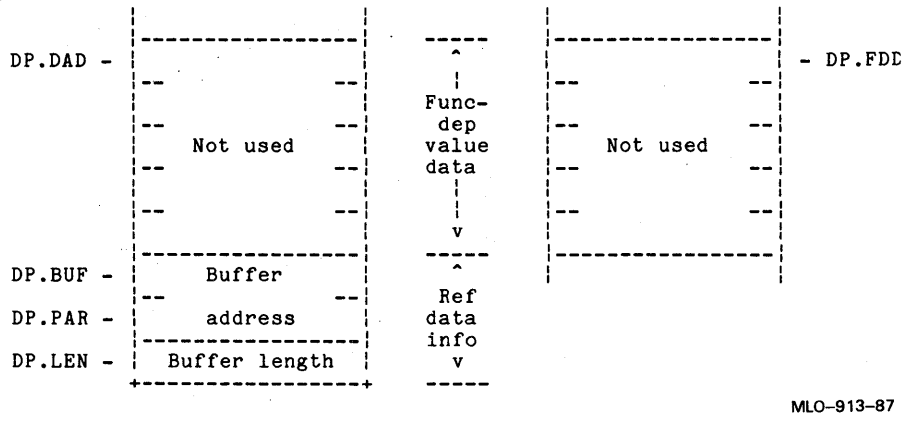
The buffer-address and buffer-length fields give the location and length, in bytes, of the buffer of command bytes.

10.5.6 Serial Poll Functions

The serial poll (IF$SPS and IF$SPO) functions are used to perform a serial poll. A serial poll is used by the controller-in-charge to determine which devices have issued a service request.

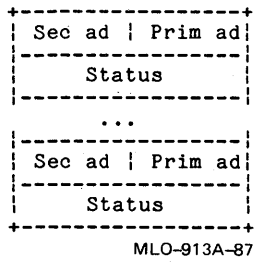
A serial poll operation begins with the deactivation of all listeners. Each device specified in the user's buffer is then addressed as "talker" for the purpose of reading its status byte. If bit 7 of a status byte is set, the currently addressed talker was requesting service. The status byte is copied to the user's buffer. If Serial Poll Over All Devices (IF$SPO) was requested, or if the service request (SRQ) bit is still set, the next device is addressed and the poll continues; otherwise, the poll terminates.

The function-dependent portions of the serial poll request and reply packets are shown below:



MLO-913-87

The buffer-address and buffer-length fields give the location and length, in bytes, of the buffer that lists the devices to be polled. The buffer format is shown below:



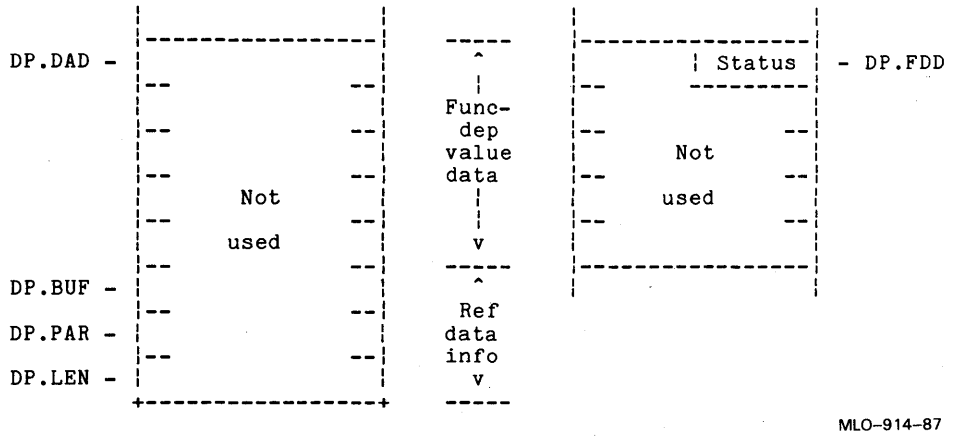
MLO-913A-87

The first word of each pair gives the talker address—with extended addressing if function modifier FM$EAD is set. The talker's status byte is returned to the second word of each pair.

10.5.7 Parallel Poll Function

The Parallel Poll (IF$PPO) function is used by the controller-in-charge to perform a parallel poll. The driver loops for 100 microseconds to wait for the parallel poll to complete. The result of a parallel poll is a status byte that is returned in the low-order byte of the first word of the function-dependent reply data. The byte contains a bit for each of eight devices selected via the Parallel Poll Configure (IF$PPC) function.

The function-dependent portions of the Parallel Poll request and reply packets are shown below:

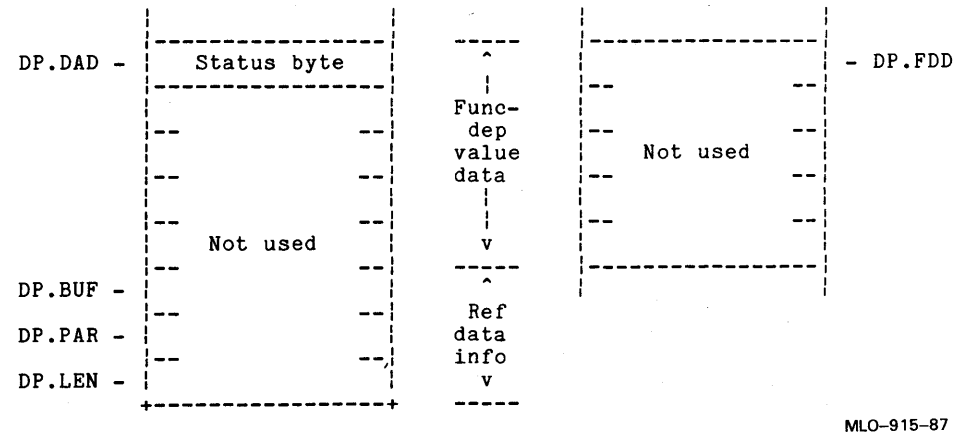


MLO-914-87

10.5.8 Load Parallel Poll Register Function

The Load Parallel Poll Register (IF$LPP) function loads a status byte into the parallel poll register. When a parallel poll is conducted, this information is transferred to the controller-in-charge. The user should update this byte to reflect the state of the device. The format of the byte is determined by a Parallel Poll Configure (IF$PPC) function request.

The function-dependent portions of the IF$LPP request and reply packets are shown below:

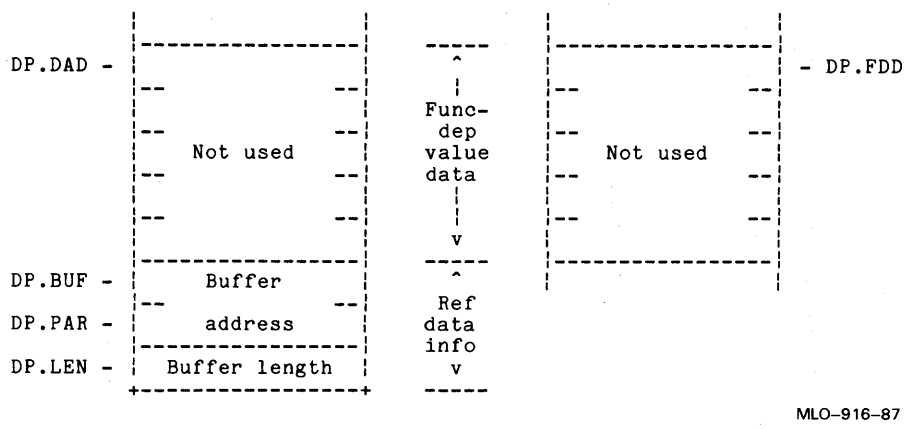


MLO-915-87

10.5.9 Parallel Poll Configure Function

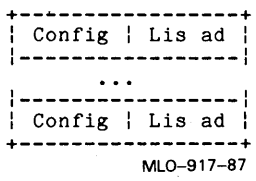
The Parallel Poll Configure (IF$PPC) function is used by the controller-in-charge to configure other devices on the IEEE bus for a parallel poll. It instructs those devices on how to respond to a parallel poll.

The function-dependent portions of the IF$PPC request and reply packets are shown below:



MLO-916-87

The buffer-address and buffer-length fields give the location and size, in bytes, of the configuration data buffer. That buffer has the format shown below:



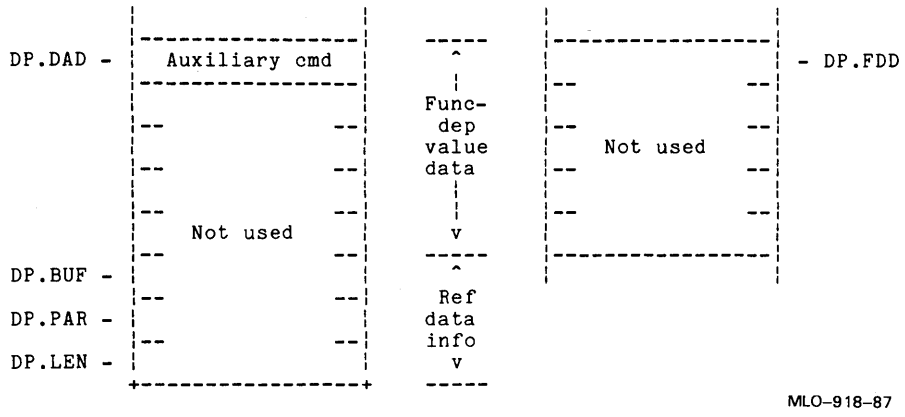
MLO-917-87

Each parameter pair in the buffer gives a listener address in the range 40 to 76 (octal) and a configuration byte with a value in the range 0 to 7. The listener address selects a device, and the configuration byte value determines which bit in the parallel poll register the device will update to reflect its state. The state of that bit is returned to the controller-in-charge when a parallel poll occurs.

10.5.10 Auxiliary Command Function

The Auxiliary Command (IF$AUX) function issues an auxiliary command by writing a byte to the auxiliary command register. Auxiliary commands are used to enable and disable most of the selectable features of IEQ11–A registers.

The function-dependent portions of the IF$AUX request and reply packets are shown below:



The auxiliary commands are described in detail in the *IEQ11–A User's Guide*. Their values are shown below:

Value	Command
0.	Software reset
1.	Release ACDS holdoff
2.	Release RFD holdoff
3.	Holdoff all data
5.	Set new byte available false (bit 7 not applicable)
8.	Assert force end or identify (FEOI) on next byte
9.	Listen only
10.	Talk only
11.	Go to standby
12.	Take control asynchronously
13.	Take control synchronously
14.	Request parallel poll
15.	Send interface clear

Value	Command
16.	Send remote enable
17.	Request control
18.	Release control
20.	Pass through next secondary
24.	Service request bit 2
Bit 7	Clear/set bit (required for clear/set type of features)

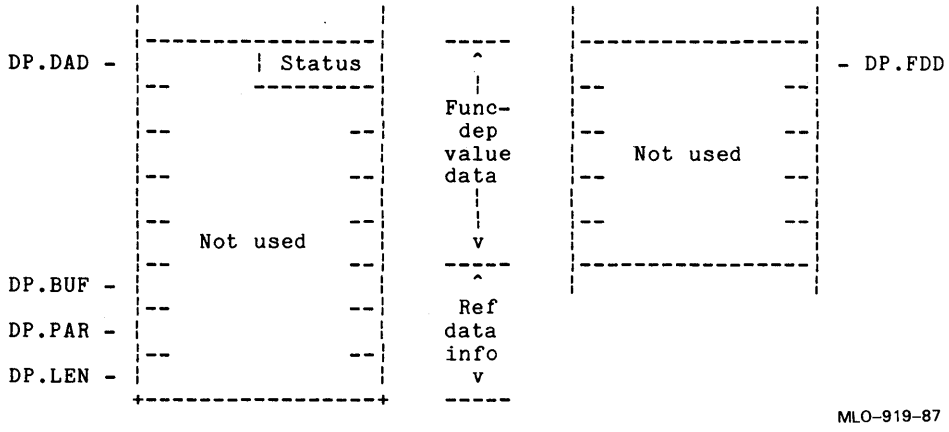
10.5.11 Request Service Function

The Request Service (IF$RSV) function generates a service request to the controller-in-charge and places a user-provided status byte in the serial poll register. The controller-in-charge will detect that the SRQ line has asserted and perform a serial poll to determine which device requested service. When the requesting device is addressed in a serial poll, the controller-in-charge will read the status byte from the serial poll register. The meaning of the status byte is application-dependent, but it should indicate the type of servicing required to the controller-in-charge.

Note

Bit 6 of the status byte is used to set the SRQ line and is not available to the user's program.

The function-dependent portions of the IF$RSV request and reply packets are shown below:



10.5.12 Get Control Function

The Get Control (IF$GTC) function is used by the controller-in-charge to go from controller standby state (CSBS) to controller active state (CACS). If the unit is already in CACS when the request is issued, no operation is performed. A state error is generated if the unit is not controller-in-charge.

Since the driver automatically goes to the state required to process a request, this request is not normally needed.

The function-dependent portions of the Get Control request and reply packets are not used.

10.5.13 Go to Standby Function

The Go to Standby (IF$GTS) function is used by the controller-in-charge to go from controller active state (CACS) to controller standby state (CSBS). If the unit is already in CSBS when the request is issued, no action is taken. A state error is generated if the unit is not controller-in-charge.

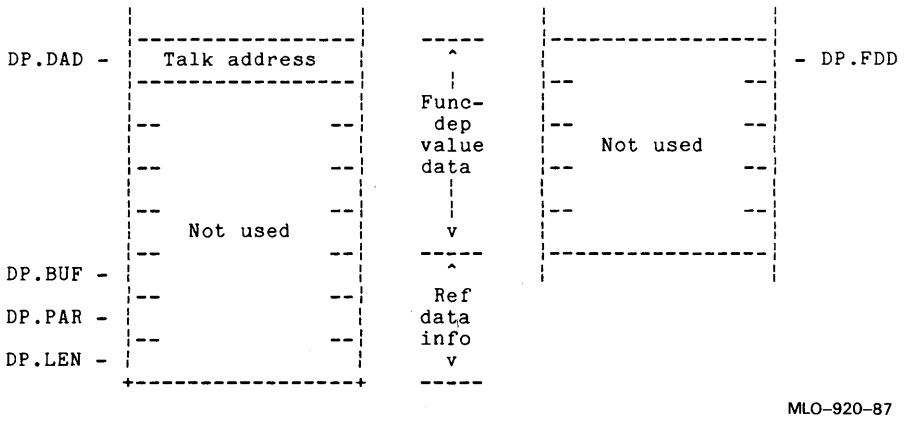
Since the driver automatically goes to the state required to process a request, this request is not normally needed.

The function-dependent portions of the Go to Standby request and reply packets are not used.

10.5.14 Pass Control Function

The Pass Control (IF$PCT) function passes control to a user-specified device, which takes control in controller active state. The device is addressed to receive control as "talker," a handshake is completed, and control is passed (by releasing the ATN line).

The function-dependent portions of the Pass Control request and reply packets are shown below:



MLO-920-87

The talker address must be in the range 100 to 136 (octal).

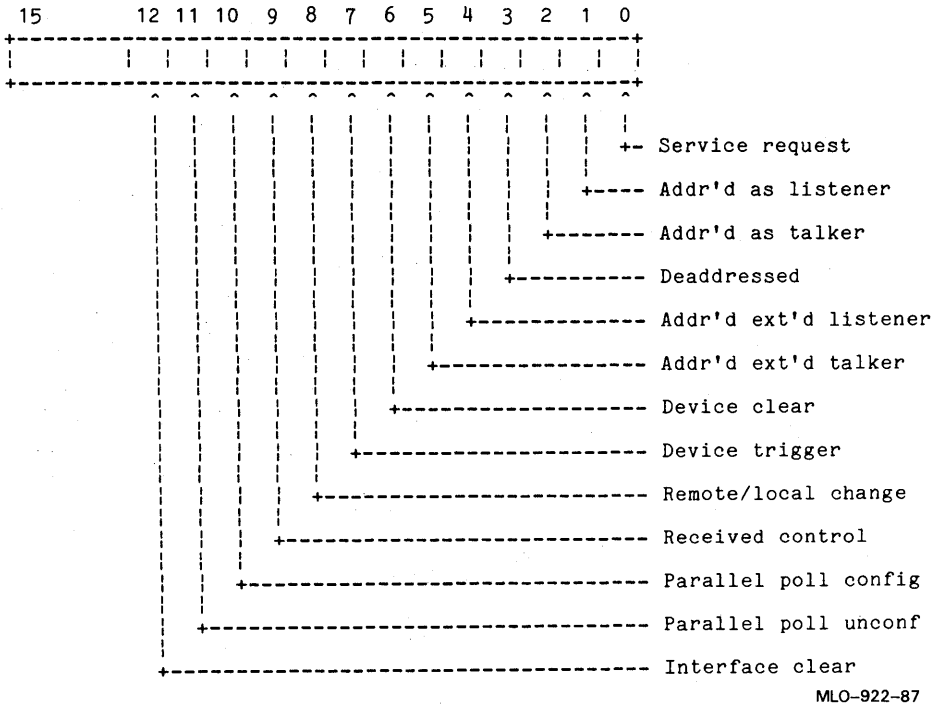
10.5.15 Set Event Mask Function

The Set Event Mask (IF$SEM) function sets up event recognition by specifying events for which recognition is requested. Those events are subsequently detected and reported to the user. (See the IF$REV and IF$RES requests.) The events are specified in a bit mask. If a bit is set in the mask, recognition of the corresponding event is requested.

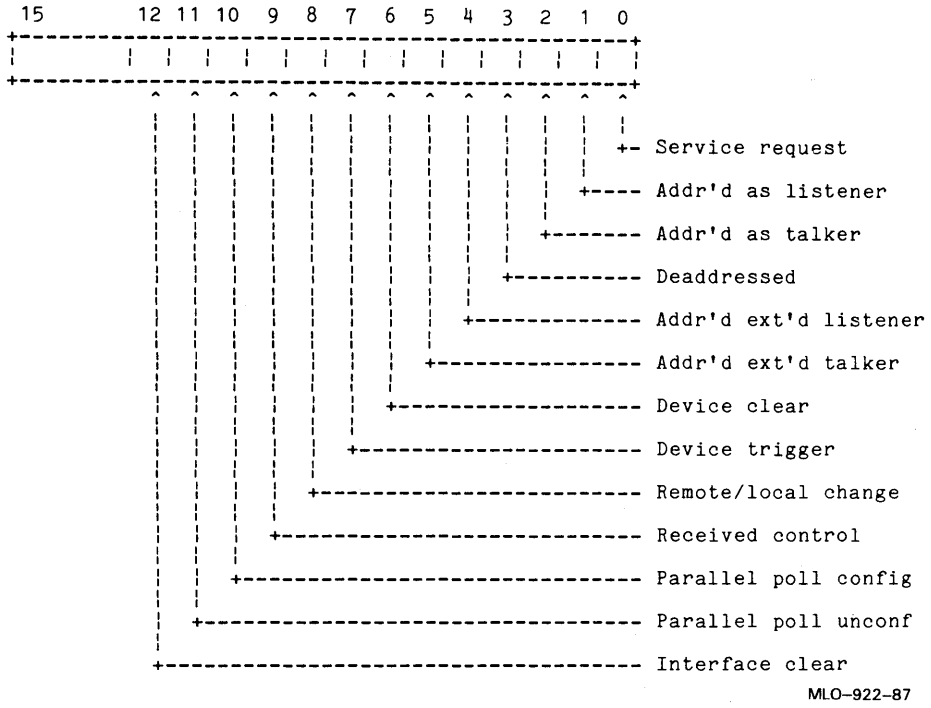
Normally the controller-in-charge requests recognition of a service request, and devices which are not controller-in-charge request recognition of all the other events.

Event recognition remains enabled until the request is issued with an argument of 0.

The function-dependent portions of the IF$SEM request and reply packets are shown below:



The event mask has the following format:

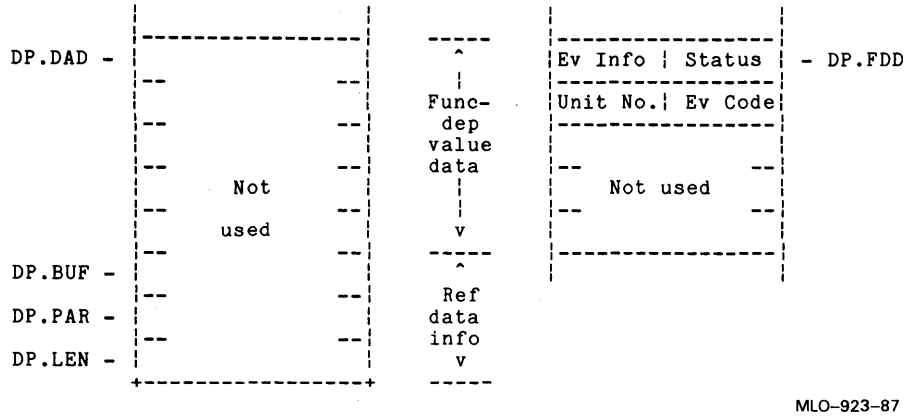


If recognition of an event represented by bits 1–9 is requested and the event occurs, the IEEE-488 is locked until released by the program. The bus can be released only by an IF$AUX request with an argument of 1 (release ACDS holdoff) or when a new request is issued to the controller, in which case the driver automatically releases the bus.

10.5.16 Wait for Event and Recognize Event Functions

The event wait and recognition functions (IF$REV and IF$RES) return notification of an event to the user. Event recognition must previously have been enabled with an IF$SEM request. If the event has already occurred, the request is returned immediately with notification. Otherwise, the request either waits until the event occurs (IF$REV) or returns with a status code indicating that no event has occurred (IF$RES).

The function-dependent portions of the IF$REV and IF$RES request and reply packets are shown below:



The status field has the value 2 when no such event has occurred for IF$RES requests; otherwise, it has the value ES$NOR (0).

The event code indicates the event that has occurred as follows:

Value	Event
1.	Service request
2.	Addressed as listener
3.	Addressed as talker
4.	Deaddressed
5.	Addressed as extended listener
6.	Addressed as extended talker
7.	Device clear
8.	Device trigger
9.	Remote/local change
10.	Received control
11.	Parallel poll configure
12.	Parallel poll unconfigure
13.	Interface clear

The event-information byte contains the following event-dependent data:

- If extended listener or talker, the extended address
- If a remote/local change occurred, 0 if the unit is in local mode, or 1 if the unit is in remote mode
- If a Parallel Poll Configure occurred, the PPE or PPD byte
- For all other events, 0

10.6 Status Codes

If an error is detected during an I/O operation by the XE driver, the driver returns an exception code in the status field of the reply message. Extended error codes are returned also; see Section 10.7.

If no error was detected, a value of ES$NOR (0) is returned in the status-code (DP.STS) field of the reply message, and a value of IE.SUC (1) is returned in the error field of the reply message.

The XE driver returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	Driver aborted
ES$CTL	HARD_IO	Controller error
ES$IVP	HARD_IO	Bad match character count for read, bad talker address for serial poll or Pass Control, bad listener address for Parallel Poll Configure, bad Set Characteristics parameter
ES$SPD	HARD_IO	SRQ or IFC terminated data transfer
ES$IFN	SOFT_IO	Illegal function code, bad controller state, event recognition already active

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for kernel-detected errors that the XE driver raises rather than passing back to the requesting process.

10.7 Extended Error Information

The XE driver returns extended error information in the error-info (DP.ERR) field of the reply message. The possible return values are listed below by symbolic and decimal error code:

Value	Meaning
-10	Data transfer terminated by IFC
-9	Bad listener address specified
-8	Bad match character count specified
-7	Bad talker address specified
-6	Event recognition already active
-5	Data transfer terminated by SRQ
-4	Error interrupt
-3	Bad controller state
-2	Bad parameter
-1	Invalid IEEE function
1	Successful completion

10.8 XE Driver Prefix File

Figure 10–1 shows the XE driver prefix module. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The DRVCF$ macro is invoked once in an XE prefix file. Its parameters are the device mnemonic (XE) and the number of configured controllers (IEQ11–A ports).

The CTRCF$ macro is invoked once for each configured controller. Its parameters are controller identifier (A, B, ...), number of units per controller (always 1), and controller CSR address and vector address.

The XE$xxx symbols define the hardware interrupt priority and the driver initialization and request-handling process priorities.

Figure 10–1: Instrument Bus Driver Prefix File (XEPFX.MAC)

```
.title   XEPFX   - IEQ Device driver prefix module
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS
; RESERVED.
;

        .mcall   drvcf$
        .mcall   ctrcf$
        .mcall   xeisz$

        xeisz$

XE$PPR  ==   175.       ; Process priority
XE$HPR  ==   4          ; IEQ hardware priority
XE$IPR  ==   250.       ; Process initialization priority

        drvcf$   dname=XE,nctrl=2
;       drvcf$   dname=XE,nctrl=1
        ctrcf$   cname=A,nunits=1,csrvec=<160150,420>,units=<0>
        ctrcf$   cname=B,nunits=1,csrvec=<160150,424>,units=<1>

        .end
```

Chapter 11
Network Service Process

This chapter describes the MicroPower/Pascal network service process (NSP), which in cooperation with the ancillary control process (ACP), communication and protocol I/O drivers, and the Pascal file system OTS (or equivalent user routines) allows a MicroPower/Pascal application to communicate and exchange data with another application residing on a remote computer.

The NSP provides two basic task-to-task facilities—MicroPower/Pascal DECnet and MicroPower/Pascal (non-DECnet) communication. With DECnet, the remote application is not restricted to MicroPower/Pascal applications but could be tasks running on VAX/VMS, VAXELN, RSX, or any other operating system that supports DECnet. Without DECnet, the remote application must be another MicroPower/Pascal process.

11.1 NSP Features and Capabilities

The NSP provides the following facilities:

- In conjunction with the ACP process, ensures the orderly establishment of task-to-task communication paths (called logical links)
- Coordinates the sequencing and flow of data between the tasks
- In conjunction with lower network layers, provides for retransmission of erroneous data received
- Performs reliability checks on the logical link to ensure that data communication paths are in working order
- Coordinates the orderly termination of logical links
- Isolates multiple logical links so that several applications can access the network simultaneously
- Informs the application of errors in establishing or manipulating the logical link

MicroPower/Pascal DECnet is an Ethernet (DEQNA) or asynchronous serial line based facility that is compatible with Digital Network Architecture (DNA) products. MicroPower/Pascal communication is a facility that connects two processes on different MicroPower/Pascal target machines by means of asynchronous or synchronous interfaces with lower overhead than MicroPower/Pascal DECnet. Both facilities use standard Pascal input/output statements for establishing logical links and data exchange.

Note

Transparent remote file access is not implemented.

11.2 Accessing the NSP for Task-to-Task Communication

For most MicroPower/Pascal applications, you access the NSP implicitly by opening a task-to-task communication path (logical link) with the Pascal OPEN statement and then issuing Pascal statements that input or output data. With both MicroPower/Pascal DECnet and MicroPower/Pascal (non-DECnet) communication, task-to-task communication is treated as just another input/output device. Programs written to communicate with other tasks use standard Pascal I/O statements. The data structure controlling access and interpretation of the data exchanged by the programs is the file variable. The NSP places no restriction on the type or content of the exchanged data other than that it must be consistent with the method of defining files in Pascal. For example, programs may use TEXT, FILE of CHAR, or FILE of INTEGER when defining the content of the messages passed between them. There is no enforced requirement that both programs use the same definition. Once the logical link between the two tasks is established, the data flow is bidirectional; either task may WRITE/PUT data to the logical link or READ/GET data from it, and eventually CLOSE the link. The synchronization of data direction is the responsibility of the two applications.

When two tasks communicate, one assumes the role of the active partner; it must identify the remote computer and the name of the task with which it will communicate. The other task takes the role of the passive partner, awaiting a request for communication from an active task. The method of specifying task names varies for each operating system. This chapter and Chapter 9 of the *MicroPower/Pascal Language Guide* describe the method used by MicroPower/Pascal. Please see the appropriate documentation for other operating systems.

Under MicroPower/Pascal, a passive task issues an OPEN statement of the form:

```
OPEN (fvar, 'SY$NET:"TASK=RECEIVER"', HISTORY:=NEW);
```

This specifies that the program is establishing itself as a passive task whose name is RECEIVER. The program will remain in a wait state until an active task initiates a connection.

The active task issues an OPEN statement of the form:

```
OPEN (fvar, 'node::"TASK=RECEIVER"', HISTORY:=OLD);
```

This specifies that the program is initiating a connection to a passive task whose name is RECEIVER, located on node "node", that is waiting to accept a connection. The syntax of the OPEN statements is compatible with VAX/VMS Pascal syntax for task-to-task communication.

A passive task that is already engaged in a dialog with another task is not eligible to accept another connection. (A file variable describes exactly one task-to-task dialog.) Since it is sometimes desirable to have multiple active tasks initiate dialogs with a common passive task, the passive task must, at the completion of the OPEN statement, spawn another process that issues the identical OPEN statement. This new task will then be available for subsequent connections.

For each MicroPower/Pascal application that engages in task-to-task communication, in addition to issuing the OPEN and subsequent I/O procedure calls, you must:

1. Edit the PROCESSOR macro in the system configuration file to specify a clock argument (for line timing) and edit the DEVICES configuration macro to reflect the clock interrupt vector address
2. Edit the NSP prefix file to indicate:
   - NSP initialization and request-handling process priorities
   - The node area, node number, and maximum segment size in particular, plus the maximum number of concurrent logical links and number of kernel packets reserved per buffer
   - The non-DECnet device(s) and/or the single DECnet device to be supported (NETDV$ macro)
3. Edit the ACP prefix file to enable network support (see Section 2.6)
4. Build into your application the following I/O system components:
   - NSP process
   - Communication and/or protocol (data-link) drivers as defined in the NSP prefix file
   - ACP process
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any support routines you opt to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for task-to-task communication, see Section 11.6, Chapters 12 and 13, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

Note

It is possible to bypass the NSP in order to access a communication or protocol (data-link) driver directly. This can be accomplished via send/receive operations to a driver's request queue semaphore or, in the case of the asynchronous DDCMP (CS) and KXT11-CA/KXJ11-CA TPR (KX/KK) drivers, by opening the device (for example, an OPEN of 'CSA0:'). Chapters 12 and 13 refer to these forms of access as "data link level (send/receive) I/O" and "nonmultiplexed MicroPower/Pascal communication," respectively.

It is also possible, given detailed knowledge of the ACP and NSP request/reply packet interfaces, to either bypass the file system OTS in order to access the ACP directly (and through it the NSP), or bypass the file system OTS and the ACP to access the NSP directly. Chapters 12 and 13 refer to such access as "low-level file system access." However, the current version of this manual does not provide detailed descriptions of the ACP and NSP send/receive interfaces.

The following sections describe the Pascal file system interface to the NSP, the NSP Set and Get Characteristics functions, the status codes that can be returned to users of the file system or request/reply packet interfaces, and the NSP prefix file. Sample programs that illustrate task-to-task communication conclude the chapter.

11.3 Pascal File System Interface

Standard PASCAL I/O statements are used to manage task-to-task communication in MicroPower/Pascal. The OPEN statement establishes logical links between active and passive tasks; the HISTORY and I/O specification parameters of the OPEN procedure are used to create active and passive links. GET, PUT, READ, and WRITE transfer data over the logical link. EOLN is valid on TEXT files over logical links and EOF is valid on any link. CLOSE and PURGE gracefully terminate logical links. BREAK and EMPTY_BUFFER terminate RECORDS in the RMS sense when sending data to operating systems that require explicit end-of-message control (for example, VAX/VMS). WRITELN performs this function automatically on TEXT files. See Chapter 9 of the *MicroPower/Pascal Language Guide* for descriptions of the Pascal I/O statements.

11.4 NSP Set and Get Characteristics Functions

The NSP supports the Set Characteristics (IF$SET) and Get Characteristics (IF$GET) standard device driver functions. The Pascal file GETSET.PAS contained in the distribution kit provides the interface routines necessary to perform those functions.

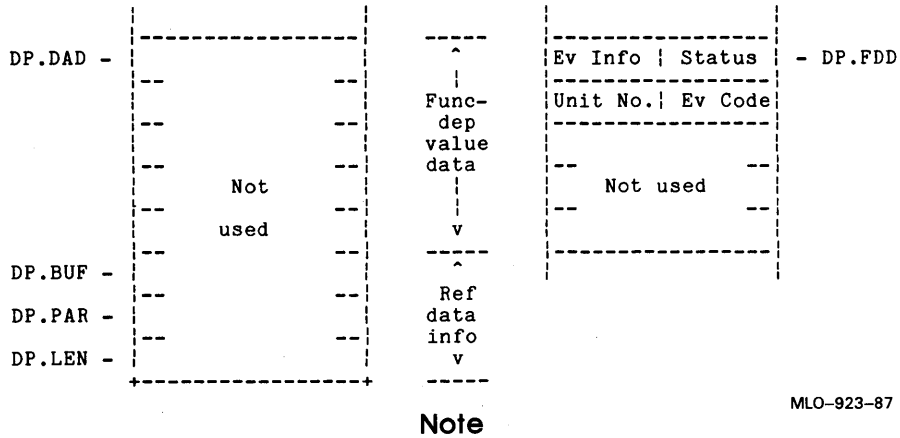
Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

11.4.1 Set Characteristics to $SECTL Queue Semaphore

You can issue a Set Characteristics request to the $SECTL queue semaphore in order to set the DECnet node number of the local computer when none was specified in the NSP prefix file or in an RSX- or VMS-host NCP command (for Ethernet downline loading).

The function-dependent portion of the NSP Set Characteristics request packet is shown below:



MLO-923-87

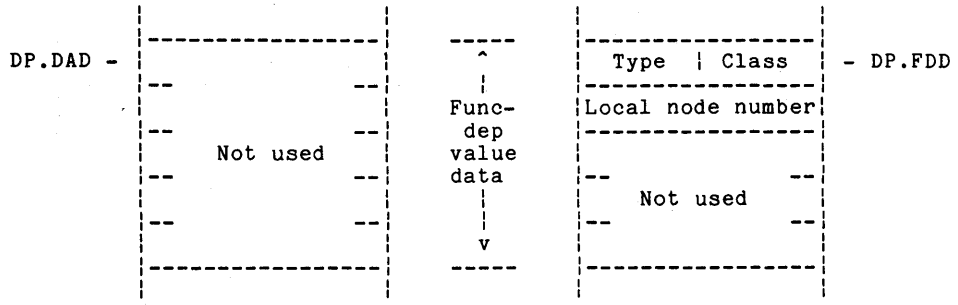
Note

The MACRO-11 field name DP.DAD, shown above, does not represent an offset into the user's send buffer; it is an offset symbol used by MACRO-11 I/O servers to reference packets. DP.DAD is a 24-byte (decimal) offset from the packet header.

11.4.2 Get Characteristics to $SECTL Queue Semaphore

You can issue a Get Characteristics request to the $SECTL queue semaphore in order to return the local system node number (or 0 if it was not specified either in the NSP prefix file or in an RSX- or VMS-host NCP command for Ethernet downline loading).

The function-dependent portions of the Get Characteristics request and reply packets are shown below:



MLO-924-87

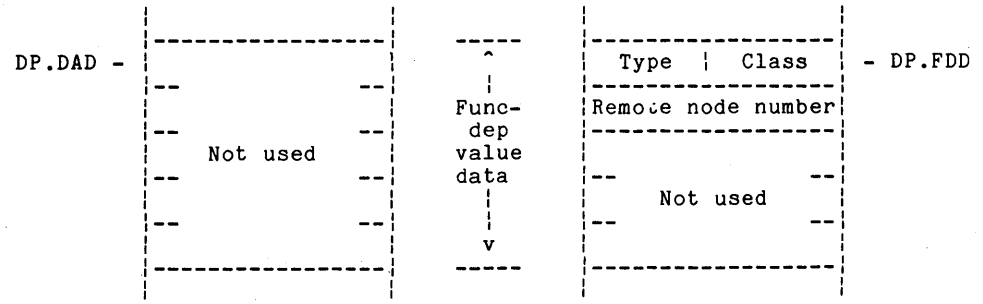
In the reply information above:

- Class is DC$SSV for system service class.
- Type is SS$NSC for network session control.

11.4.3 Get Characteristics to File Variable

You can use the module GETSET.PAS and the Pascal include file GSINC.PAS to issue a Get Characteristics request to the logical-link file variable, in order to return the node number of the remote partner.

In this case, the function-dependent portions of the Get Characteristics request and reply packets are:



MLO-925-87

In the reply information above:

- Class is DC$SSV for system service class.
- Type is SS$NLL for network link server.

11.5 Status Codes

Any error detected by the NSP during an I/O operation is returned in the status-code field of a reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements—the Pascal OTS will report the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). The error codes listed in this section are those generated by the NSP directly—not those generated by other I/O system components involved in task-to-task data exchange.

If no error is detected during the I/O operation, the NSP returns a value of ES$NOR (0) in the status-code field.

The NSP returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O request canceled or aborted. This status is returned when the remote partner purges the logical link. PURGE normally indicates an abnormal termination of the logical link.
ES$DAL	HARD_IO	Device already allocated. This status is returned when an attempt to set the local node number with Set Characteristics is made and the local node number is already defined.
ES$OVF	HARD_IO	Data buffer overflow.
ES$DNU	SOFT_IO	Destination node is unreachable. This status indicates that the node requested by an active task is not currently reachable via the network.
ES$EOF	SOFT_IO	End of file. This status is returned when the remote partner closes the logical link. CLOSE is the normal method of termination of a logical link. The Pascal OTS interprets this status and causes the EOF function to return TRUE.
ES$IFN	SOFT_IO	Illegal function.
ES$IVL	SOFT_IO	Invalid length.
ES$LRJ	SOFT_IO	Link rejected by the remote task. This status indicates that the remote passive task exists but has decided not to accept the logical link. This error cannot occur if the remote computer is another MicroPower/Pascal target.
ES$NRF	SOFT_IO	No reference data present.
ES$PAL	SOFT_IO	Path to remote partner lost. This status indicates that, although the remote partner was at some time connected to the logical link, there is no longer a network path to the remote computer.

Code	Type	Description
ES$TNF	SOFT_IO	Task not found. This status indicates that the node requested by the active task does not contain the desired passive task.
ES$LNR	RESOURCE	Local network resource failure. This status is returned when there are already NS$NLL logical links in use and an attempt to open another is made. If more concurrent logical links are required, increase the NS$NLL value in the NSP prefix file.
ES$RNR	RESOURCE	No resources at the remote computer. This status indicates that the node requested by the active task does not have sufficient resources to create a logical link. This error cannot occur if the remote computer is another MicroPower/Pascal target.

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU and COMM macro libraries for MACRO-11 users.

11.6 NSP Prefix File

The NSP prefix module is shown in Figure 11-1. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The modifiable constants and their default values (in decimal) are:

Symbol	Default	Description
NS$IPR	250	The priority of the NSP initialization procedure.
NS$PPR	175	The priority normally used for the NSP process. DIGITAL recommends that the priorities of the ACP, the NSP, and any driver used by the NSP be the same to avoid excessive context switching.

Symbol	Default	Description
NS$SGZ	256	The maximum individual data message size. This value is the size of the largest single message that the NSP will transmit to a remote application. User file variables may be larger (see the sample programs below) but the Pascal I/O system and the NSP will transmit the larger message as a sequence of smaller segments. The application may also request that smaller messages be transmitted by specifying the BUFFERSIZE parameter on the OPEN statement. Finally, the actual data message segment size used is the minimum of the BUFFERSIZE parameters specified by both application program OPEN statements (excluding 0) and the NS$SGZ parameter of the NSP processes on each computer. By adjusting this parameter and using the BUFFERSIZE parameter of OPEN, the application can trade off RAM usage and network message traffic.
NS$NLL	4	The maximum number of concurrent logical links. Each OPEN statement consumes one of these logical links, and each CLOSE or PURGE releases one.
NS$XKP	2	The number of kernel packets reserved to the NSP process in addition to one per communication buffer. If kernel packets are a scarce resource in the application, reserving some to the NSP will increase network performance.
NS$DNN	0	The DECnet node number. For MicroPower/Pascal DECnet, each node must have a node number. This value can be supplied in the prefix file in an RSX- or VMS-host NCP command (for Ethernet down-line loading) or in a Set Characteristics request. The NCP value applies for a mapped application that is configured to respond to a network boot request (SYSTEM debug=NO, netboot=YES). Otherwise, the prefix file value applies. If network booting is disabled (SYSTEM netboot=NO) and 0 (zero) is specified in the prefix file, the NSP process does not participate in DECnet until a node number is defined using the Set Characteristics function.
NS$DNA	0	The DECnet area number. The formula NS$DNA * 1024. + NS$DNN defines the actual Phase IV DECnet node address.

Communication devices used by the NSP fall into two categories—DECnet and MicroPower/Pascal-only. DECnet devices are the QN driver (type=ETHER) for the DEQNA Ethernet interface and the CS driver (type=POINT) for the asynchronous DDCMP interface. Only one of these may be specified in the prefix file. MicroPower/Pascal-only devices (type=UPOWER) are the CS driver, the XP driver for the DPV11, the XS driver for the synchronous port of the KXT11-CA/KXJ11-CA, and the KX/KK drivers for the arbiter-KXT11-CA/KXJ11-CA communication port.

Figure 11–1: NSP Prefix File (NSPPFX.MAC)

```
        .title   NSPPFX - Network Services Process prefix module
;+
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1984, 1986 BY DIGITAL EQUIPMENT CORPORATION.
;       ALL RIGHTS RESERVED.
;-
         .mcall   macdf$,misdf$,nspdf$
         macdf$
         misdf$
         nspdf$

;+
; User changable symbols for the NSP process
;-

NS$IPR   ==   250.    ;Initialization priority
NS$PPR   ==   175.    ;Normal process priority
NS$SGZ   ==   256.    ;Maximum data message size
NS$NLL   ==   4.    ;Maximum number of concurrent logical links
NS$XKP   ==   2.    ;Packets reserved in addition to one per buffer
NS$DNN   ==   0.    ;DECnet node number (Unspecified)
NS$DNA   ==   0.    ;DECnet area number (Unspecified)

;+
; The definition of the communication devices
;
; For MicroPower-to-MicroPower devices:
;
;   NETDV$ QSEM=<ssssss>,UNIT=u,NUMBFR=b,TYPE=UPOWER,AREA=a,ADDRESS=n
;         or
;   NETDV$ TYPE=SELF    ;To enable links to the local node (0::)
;
; For DECnet devices
;
;   NETDV$ QSEM=<ssssss>,UNIT=u,NUMBFR=b,TYPE=ETHER    ;Ethernet
;         or
;   NETDV$ QSEM=<ssssss>,UNIT=u,NUMBFR=b,TYPE=POINT    ;Async DDCMP

;
;   ssssss is a 6 character, upper case, blank filled device driver
;      named semaphore enclosed in angle brackets (e.g. QSEM=<$QNA  >).
;      This parameter must be specified, there is no default.
;
;   u       is the unit number if the device driver is capable of
;           supporting multiple units.  The default is 0 (zero).
;
;   b       is the number of input buffers dedicated to this device.  The
;           number of buffers to use is dependent on the speed of the
;           device. The default is 4 (four) but can be set to 1 for low
;           speed lines and higher for broadcast lines.  The number of
;           buffers greatly affects the RAM usage of the NSP process.
;           Each buffer also causes a kernel packet to be reserved to the NSP.
```

```
;
; The following parameters must be specified on Non-DECnet devices and
; must NOT be specified on DECnet devices.
;
;   a       is the area number associated with the node.  If no area is
;           specified, the area of the local node is used.
;
;   n       is the address within the area associated with the node.
;
; If the node specified in the PASCAL OPEN statement is associated
; with a MicroPower-to-MicroPower device then that device will be
; used, otherwise it is assumed that the node can be reached by
; the DECnet partner (designated router on Ethernet).  There is no
; requirement that the MicroPower node on the other side of the
; Non-DECnet device have NS$DNN = n and NS$DNA = a.  These
; parameters are used solely to differentiate between DECnet and
; MicroPower intersystem environments.
;
; Note 1: Any Non-DECnet devices must be specified first.
; Note 2: There can be only 1 DECnet device specified.
;-

;   NETDV$   qsem=<$CSA  >,unit=1,type=UPOWER,address=1023.
;   NETDV$   type=SELF
;   NETDV$   qsem=<$CSA  >,unit=0,type=POINT
    NETDV$   qsem=<$QNA  >,type=ETHER

  nsfin$   ;Generate the data structures
  .end     ;and all done
```

11.7 Sample Programs

11.7.1 Transferring Data Between Two MicroPower/Pascal Nodes

The following two programs exchange data. The first is the active task that continually copies and displays TEXT data from the passive task. The second is the passive task. Note that the second program spawns an identical copy of the server process upon completion of the OPEN statement. This is to ensure that there is always a passive task available.

```
[System(MicroPower)] program network_read;

var
    f : text;
    ch : char;
```

```
    begin
      while true do
        begin
          open (f, '5.410::"TASK=IDENTIFY"', BUFFERSIZE := 80
               ,HISTORY := OLD, OVERLAPPED := ENABLE);
          reset (f);
          while not eof (f) do
            begin
              while not eoln (f) do
                begin
                  output^ := f^;
                  put(output);
                  get(f);
                end;
              writeln (output);
              get(f);
            end;
          close (f);
        end;
    end.
```

```
[System(MicroPower)] program network_server;

  [stack_size(300)] process identify(year : integer);
  var
      f : text;
    begin
      open(f,'SY$NET:'(TASK=IDENTIFY)',history:=new,buffersize:=80,
           overlapped:=enable);
      identify(year+1,relation:=independent);
      rewrite(f);
      writeln(f,'Year of the city ',year:1);
      writeln(f,'Welcome stranger');
      write(f,'You have contacted Logan 5 on the node UPOWER');
      writeln(f);
      writeln(f);
      writeln(f,'Things are pretty good here except no task may exist');
      writeln(f,'for more that 30 milliseconds.  Gotta run now.');
      writeln(f);
      writeln(f,'Remember: On the net, there is no Sanctuary.');
      close(f);
    end;

  begin
    identify(1);
  end.
```

11.7.2 Transferring Data Between MicroPower/Pascal and VAX/VMS Nodes

The following programs demonstrate data exchange between a MicroPower/Pascal program/process and a VMS program. The method of data exchange is task-to-task communication. The VMS examples conform to the guidelines set forth in the *VMS DECnet User's Guide* and those referenced in the *VMS PASCAL Reference Guide*. For methods of invoking a task in other languages or on other operating systems, see the documents for those languages or systems.

To send a record of data from a MicroPower/Pascal process to a VMS program, create a command file in the appropriate VMS directory that will invoke the VMS passive task when the MicroPower task requests it.

```
$! This is command file TV.COM
$
$RUN TV      !Assumes TV.EXE is in the same directory
```

This is the VMS PASCAL program that receives the data.

```
PROGRAM TV(INPUT,OUTPUT,F);
CONST BIG_ARRAY_SIZE = 1024;
TYPE BIG_ARRAY = ARRAY [1..BIG_ARRAY_SIZE] OF INTEGER;
VAR F : FILE OF BIG_ARRAY;
    J : INTEGER;
BEGIN
  OPEN (F,'SYS$NET',HISTORY:=OLD,ACCESS_METHOD:=SEQUENTIAL);
  RESET(F);
  FOR J := 1 TO BIG_ARRAY_SIZE DO
    BEGIN
      WRITE(OUTPUT,F^[J]);
      IF F^[J] <> J
        THEN WRITE(' <----Error, should have been ',j:1);
      WRITELN;
    END;
  CLOSE(F);
END.
```

This is the MicroPower/Pascal program that constructs the data.

```
[system(micropower)] program TM(input,output);
const
    big_array_size = 1024;
type
    big_array = array [1..big_array_size] of long_integer;
var
    f : file of big_array;
    j : integer;
begin
    open (f,'5.69"(USER PASSWORD)::"(TASK=TV)"',HISTORY:=OLD);
    rewrite(f);
    for j := 1 to big_array_size do f^[j] := j;
    put(f);
    break(f);
    close(f);
end.
```

In the example above, node 5.69:: is assumed to be the DECnet node number of the VMS system, [USER] is the directory containing the TV.COM file, and PASSWORD is the valid password for the USER. USER and PASSWORD (and the "characters" can be omitted to indicate to VMS that the default DECnet account should be used.

This program uses a 4096-byte RECORD, which is the largest record that the VMS Pascal run-time library will allow. For this example, the RECORD is simply an array of 4-byte integers (VAX INTEGER is 4 bytes long, which is the size of a MicroPower/Pascal LONG_INTEGER). As long as the records are of compatible types, contents, and length, any record could be exchanged. Files of type TEXT can also be used to utilize variable-line-length ASCII data.

The MicroPower/Pascal program above was installed in a kernel image that contained the QNA (QNPFX) driver, the ACP (ACPPFX) process, and the NSP (NSPPFX) process. The NSPPFX file was edited to assign the appropriate MicroPower/Pascal node identification (node number).

Using the same sample programs as above, the following code transfers data to a MicroPower/Pascal task from a VMS image.

This is the VMS PASCAL program that constructs and sends the data:

```
PROGRAM TV(INPUT,OUTPUT,F);
CONST BIG_ARRAY_SIZE = 1024;
TYPE BIG_ARRAY = ARRAY [1..BIG_ARRAY_SIZE] OF INTEGER;
VAR F : FILE OF BIG_ARRAY;
    J : INTEGER;
BEGIN
  OPEN (F,'UPOWER::"(TASK=TM)"',HISTORY:=NEW,
        ACCESS_METHOD:=SEQUENTIAL);
  REWRITE(F);
  FOR J := 1 TO BIG_ARRAY_SIZE DO F^[J] := J;
  PUT(F);
  CLOSE(F);
END.
```

This is the MicroPower/Pascal program that receives the data:

```
[system(micropower)] program TM(input,output);
const
    big_array_size = 1024;
type
    big_array = array [1..big_array_size] of long_integer;
var
    f : file of big_array;
    j : integer;

begin
      open (f,'SY$NET:"(TASK=TM)',history:=new);
  reset(f);
  for j := 1 to big_array_size do
    begin
      write(output,f^[j]);
      if f^[j] <> j
        then write(' <----Error, should have been ',j:1);
      writeln;
    end;
  close(f);
end.
```

In the example above, node UPOWER:: is assumed to be the DECnet node name of the MicroPower/Pascal system. It is also assumed that the system manager for VMS has used NCP to set the correspondence of the node name UPOWER with the appropriate node number and that that node number corresponds to the one specified in the NSP prefix (NSPPFX) file. The VMS image can then be invoked by the running of it interactively or in batch.

The examples process the data, which shows the method of data exchange. The VMS sending program could have taken its data from a disk file and presented it to the MicroPower/Pascal process. This could be a method of transferring files between the systems, assuming some knowledge of the file names, contents, record formats, and record-blocking used by each system. Files created on VMS are truly readable only by RMS. However, if the two programs agree to exchange the data at the record level, the VMS Pascal program could read records from the file (letting RMS worry about file format) and send compatible records to the MicroPower/Pascal process. Then, the data exchange would be at the record level and would be independent of the file characteristics.

11.7.3 Determining and Setting the Local Node Number

The following program determines the local node number and shows one way to set the node number if it was not specified in the NSP prefix file or in an RSX- or VMS-host NCP command (for Ethernet down-line loading).

```
[system(micropower)] program nettst(input,output);

  %include 'GSINC'
var
  local_node : unsigned;
  main_sts : array [1..2] of unsigned;
  main_net : queue_semaphore_desc;

  function ask_node(
        var prompt:[readonly] packed array [1..h:integer] of char
                    ) : unsigned;
  var
      i : integer;
      a,n : unsigned;
      c : char;
    begin
      for i := 1 to h do write(prompt[i]);
      write(' (area.number) ? ');
      readln(a,c,n);
      ask_node := n + (a * 1024);
    end;
```

```
begin
      init_structure_desc(desc := main_net, name := '$SECTL');
      if not Get_Desc_Characteristics(main_net,0,main_sts,size(main_sts))
        then main_sts[2] := 0;
      if main_sts[2] <> 0
        then
          begin
            main_sts[1] := main_sts[2] div 1024;
            main_sts[2] := uand(main_sts[2],1023);
            writeln('Local node is ',main_sts[1]:1,'.',main_sts[2]:1);
          end
        else
          begin
            main_sts[2] := ask_node('What is the local node number');
            if Set_Desc_Characteristics(main_net,0,main_sts,size(main_sts))
              then ;
          end;
      local_node := main_sts[2];
    end.
```

Chapter 12
Asynchronous DDCMP Driver

This chapter describes the use of the MicroPower/Pascal asynchronous DDCMP (CS) driver, which implements the Digital Data Communications Message Protocol (DDCMP) for message exchange over asynchronous serial lines.

The CS driver controls the transmission of data grouped into physical blocks (data messages) over an asynchronous serial data link, while ensuring correct sequencing and integrity of the messages. Those messages are exchanged with a partner program on a different computer that also adheres to the DDCMP protocol.

Once the data is correctly exchanged, it is the responsibility of higher levels of software—for example, the MicroPower/Pascal network service process (NSP) or a user process—to interpret the exchanged data. The calling process must agree with the partner software on the format and meaning of the transmitted data. The transmitted data may, for example, contain control information that allows the higher-level software to provide such services as message segmentation or multiplexing of that exchange with other task-to-task "conversations" over the serial line.

The CS driver differs from other MicroPower/Pascal drivers in that it controls the hardware not directly but by issuing requests to another driver—the asynchronous serial line (TT) driver. For that reason, the MicroPower/Pascal software classifies CS as a "protocol" device driver, whereas the TT driver (Chapter 3) and the communication drivers (Chapter 13) are given hardware-based classifications. The TT driver provides a basic byte transmission capability; the CS driver adds message framing, message sequencing, error detection, and retransmission of messages received in error.

The CS driver supports DDCMP message exchange via the following asynchronous serial line interfaces:

- DLV11-type—DLV11, DLV11-E, DLV11-F, DLV11-J
- DLART-type—KXT11-CA/KXJ11-CA console, SBC-11/21, MXV11-A, MXV11-B
- DZV11
- DHV11
- KXT11-CA/KXJ11-CA multiprotocol chip

When used in conjunction with the CS and TT drivers, the supported devices interface one or more asynchronous serial lines to a MicroPower/Pascal target processor for communication with other processors.

MicroPower/Pascal supports three distinct levels of communication device I/O:

- MicroPower/Pascal DECnet
- MicroPower/Pascal Communication
- Data link level (send/receive) I/O

Section 12.2 describes each level of communication device access and summarizes the possible paths through the CS driver.

12.1 CS Driver Features and Capabilities

The CS driver provides a standard driver interface to DDCMP, a byte-oriented data link protocol. The MicroPower/Pascal implementation of DDCMP provides sequential, error-free message delivery over an asynchronous serial communication link. The protocol performs the following functions:

- Message framing: constructs or interprets DDCMP data messages and control messages (described below)
- Error detection: detects errors in message headers or data via cyclic redundancy checksum (CRC) checks
- Retransmission of messages received in error
- Message sequencing: numbers messages in order to prevent duplications or omissions and to identify retransmissions

In a DDCMP data message, data to be transmitted is preceded by a header that includes a special beginning character, a byte count that indicates the length of the data portion (up to 16383 bytes), and control information. The control information can include the number of the last correctly received message (piggybacked acknowledgment) and the current packet's sequence number. CRCs follow both the header and data portions.

In addition to the data messages, there are five control messages: ACK, NAK, REP, START, and STACK. ACK conveys acknowledgment of successful message receipt when there is no reverse traffic onto which to piggyback a response. NAK carries notification of an error and its cause, while implicitly acknowledging successful receipt of all previous messages. REP is sent when a transmitter times out waiting for acknowledgment; it requires an ACK or NAK response. Also, in the MicroPower/Pascal implementation, REP is sent at user-defined intervals as a "keep-alive" timing mechanism (unless the calling process—for example, the NSP—is timing the line). START and STACK initialize a DDCMP link.

The DDCMP protocol can accommodate many different methods and modes of transmission. The MicroPower/Pascal implementation uses full duplex transmission in asynchronous serial mode.

Note

MicroPower/Pascal DDCMP is for point-to-point use only. In the data message, the select flag (used in half-duplex or multipoint configurations to

turn transmitters on and off) is always 0, and the station address (used to identify multipoint tributaries) is always 1.

Also, MicroPower/Pascal DDCMP does not support maintenance mode and the messages associated with it.

The CS driver supports read and write operations, protocol enabling or disabling, and the returning of "device" characteristics. Indirect reference pointers are honored on write operations for the purpose of performing gathered writes into a DDCMP data message.

The CS driver can be accessed by the NSP (which normally implies filesystem OTS and ACP involvement) in connection with setting up logical links and multiplexing them across physical links (virtual circuits) for task-to-task I/O. The driver can also be opened for Pascal file I/O without NSP involvement (for example, an OPEN of 'CSA0:'), or used directly by a user process for data link level (send/receive) I/O.

The NSP uses the CS driver for the following types of task-to-task communication:

- DECnet endnode support, for communication between a MicroPower/Pascal target and another DECnet node—possibly another MicroPower/Pascal target, a VAX/VMS or RSX system, or other system running DECnet
- Asynchronous point-to-point communication between two MicroPower/Pascal targets

12.2 Performing Asynchronous DDCMP I/O

MicroPower/Pascal supports three distinct levels of communication device I/O:

- MicroPower/Pascal DECnet
- MicroPower/Pascal communication
- Data link level (send/receive) I/O

For most MicroPower/Pascal applications, asynchronous DDCMP message exchange is performed with the MicroPower/Pascal DECnet or MicroPower/Pascal communication facilities.

MicroPower/Pascal DECnet is an asynchronous serial line-based or Ethernet-based (DEQNA) facility that is compatible with Digital Network Architecture (DNA) products. Using MicroPower/Pascal DECnet, a MicroPower/Pascal target machine may communicate with processes in other MicroPower/Pascal targets, with processes in VAXELN targets, or with tasks in VAX/VMS, RSX, or other systems.

MicroPower/Pascal communication allows the user to exchange data between processes on different MicroPower/Pascal target machines by means of standard Pascal Input/Output statements.

Note

Transparent remote file access is not supported. One method for transferring files between systems, using task-to-task data exchange, is noted in Chapter 11.

Both the MicroPower/Pascal DECnet and communication facilities provide sequential, error-free data delivery, while hiding the details of data exchange, such as initialization and error recovery. Both facilities can carry on many task-to-task dialogs across a single physical link. The physical link is controlled to ensure that there is no "crosstalk" between the multiplexed logical links.

The MicroPower/Pascal DECnet and Communication facilities have the following components:

- The NSP coordinates the flow of data between two processes. The NSP conforms to the DNA specifications for the Session Control layer, the End Communication (Network Services) layer, and the Routing layer of DECnet. See Chapter 11 for details.
- The CS driver monitors the data transfer between two MicroPower/Pascal target machines and performs the appropriate recovery algorithms to correct transmission errors. The CS driver conforms to the DNA specification for DDCMP and the DECnet Data Link layer. (CS is a protocol driver and does not control hardware directly; it uses the TT driver to perform lower-level data link functions.)
- The communication drivers (QN, XP, XS, KX, and KK) control communication device hardware, performing such data link functions as message framing and error detection. See Chapter 13 for details.

Both MicroPower/Pascal DECnet and Communication treat task-to-task communication as a normal input/output device. Programs written to communicate with other tasks use standard Pascal I/O statements. The data structure controlling access and interpretation of the data exchanged by the programs is the file variable. The OPEN statement establishes logical links between both active and passive tasks; the HISTORY and I/O specification arguments to the OPEN procedure create active and passive links. GET, PUT, READ, and WRITE transfer data over the logical link. EOLN is valid on text files over logical links, and EOF is valid on any link. CLOSE and PURGE are used to terminate logical links gracefully. Chapter 9 of the *MicroPower/Pascal Language Guide* describes the exact syntax of the Pascal I/O statements.

Note

When multiplexing of logical links over a CS serial line is not necessary, you can eliminate the NSP from your application and open the serial line by specifying "CSA0:" (for example) in the OPEN statement. Nonmultiplexing is a special case of MicroPower/Pascal Communication and applies only to protocol-class and communication-class drivers that allow direct opens—CS, KX, and KK.

For each MicroPower/Pascal application participating in DDCMP-based data exchange, in addition to invoking the Pascal I/O procedures, you must:

1. Edit the PROCESSOR macro in the system configuration file to specify a clock argument (for line timing) and edit the DEVICES configuration macro to reflect the serial line controller and clock interrupt vector addresses
2. Edit the TT driver prefix file to reflect:
   - [For each controller:] Controller type, CSR address, interrupt vector address, hardware interrupt priority, and number of serial lines
   - [For each line:] ISR buffer size and line speed; XON/XOFF flow control and line editing must be disabled
   - Driver initialization and request-handling process priorities
3. Edit the CS driver prefix file to reflect:
   - [For each serial line:] TT queue semaphore identifier and TT unit number; the CS unit number will normally NOT correspond to the TT unit number

- Other interface characteristics, such as the ACK timeout interval and (if the calling process is not timing the line) the interval between "keep-alive" message transmissions
- Driver initialization and request-handling process priorities

4. [For logical link OPEN:] Edit the NSP prefix file to define the communication devices available to the NSP process. Non-DECnet devices precede a DECnet device, and only one DECnet device may be specified. Specify TYPE parameter POINT for DECnet DDCMP. For DDCMP Communication (non-DECnet), specify type UPOWER and supply an address (parameters AREA and ADDRESS).
5. [For OPEN:] Edit the ACP prefix file to indicate whether NSP open support is required; the default is inclusion of NSP open support. Also, check the ACP pool size; 180 bytes are required per NSP open.
6. Build into each participating MicroPower/Pascal application the following I/O system components:
   - TT driver process
   - CS driver process
   - [For logical link OPEN:] The NSP
   - [For OPEN:] The ACP
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any I/O support routines you choose to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for DDCMP protocol I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 12.6 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

When a module that contains Pascal I/O procedure invocations is built into your application, Pascal OTS routines for file service are linked to the module. The OTS file routines perform all Pascal operations on logical-link or device files, including opening, input, and output. In particular, they perform the necessary low-level processing of high-level operations, such as OPEN and WRITE. Thus, the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to driver or NSP or ACP queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores—are concealed from the Pascal user.

As alternatives to using the MicroPower/Pascal DECnet or Communication facilities for DDCMP protocol I/O, you can:

- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and an NSP logical-link server (or CS driver if no NSP), bypassing the OTS file routines (lower-level file system access)
- Issue your own Pascal or MACRO–11 packet-level requests to the CS driver, bypassing the OTS file routines, the ACP, and the NSP (low-level nonfile access)

Table 12–1 summarizes the possible paths through the CS driver. For reference, the three levels of communication represented are MicroPower/Pascal DECnet, MicroPower/Pascal Communication, and data link level (send/receive) I/O.

Table 12–1: Asynchronous DDCMP I/O Paths and Interfaces

Type of	I/O System Components			Interface	
Communication	**ACP/NSP**	**Prot**	**Comm/TT**	**Pascal**	**MACRO-11**
DECnet/DDCMP or DDCMP Communication	ACP/NSP	CS	TT	OPEN link	SEND$ to ACP (or NSP using ACP format)
DDCMP nonmux Communication	ACP	CS	TT	OPEN comm. line	SEND$ to ACP
DDCMP (data link level)	—	CS	TT	SEND to driver	SEND$ to driver

The following sections describe the Pascal I/O statement interfaces to the CS driver, the lower-level request/reply packet interface, the status codes that can be returned to users of any interface, and the CS driver prefix file.

12.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O to a DDCMP communication line, you issue an OPEN statement that associates a file variable with a logical link. (An alternative—the direct opening of a communication line—is addressed below.) The file variable controls access and interpretation of data as it is exchanged, via the serial line, with a cooperating program on a different machine. No restriction is placed on the type or contents of the exchanged data, as long as all is consistent with the method of defining files in Pascal. Programs may use TEXT, FILE of CHAR, FILE of INTEGER, and so forth when defining the content of the messages passed between them. Nor must both programs use the same definition.

To accomplish the dialog between tasks, one task must take the role of the initiator, or active task, while the other defines itself as the target, or passive task. The passive task must define a name that identifies the task to an active task; the active task specifies the name of the passive task when initiating the connection. Task names can be up to 16 characters long. When initiating the connection, the active task must specify the machine containing the passive task it is attempting to locate. MicroPower/Pascal DECnet will dynamically determine the location of each named machine. For MicroPower/Pascal Communication, each serial line is associated with a static name that identifies the machine to which it is connected.

Once the connection between the two tasks is established, the dialog is bidirectional; that is, either task may WRITE/PUT data to it or READ/GET data from it, and eventually close it, as if it were a file residing on a local device. The synchronization of data direction is the responsibility of the two programs.

For example, a passive task can issue an OPEN statement of the form:

```
OPEN (fvar, 'SY$NET:"TASK=taskname"', HISTORY:=NEW);
```

This specifies that the program is establishing itself as a passive task with the name "taskname". The program will remain in a wait state until an active task initiates a connection.

The active task issues an OPEN statement of the form:

```
OPEN (fvar, 'node::"TASK=taskname"', HISTORY:=OLD);
```

This specifies that the program is initiating a connection to a passive task named "taskname" located on node "node" that is waiting to accept a connection. The syntax of the OPEN is compatible with VAX/VMS Pascal syntax for task-to-task communication.

A passive task that is already engaged in a dialog with another task is ineligible to accept another connection. (A file variable describes exactly one task-to-task dialog.) If you want multiple active tasks to initiate dialogs with a common passive task, the passive task must, at the completion of the OPEN statement, spawn another process that issues the identical OPEN statement. The new task will then be available for subsequent connections.

The preceding discussion of the OPEN statement applies to MicroPower/Pascal DECnet and to MicroPower/Pascal Communication in the case where several "conversations" are occurring on the data channel. However, if the serial line is to be used by a single process in one target talking to a single process in another target, you can avoid the overhead of the NSP process simply by not installing it into the application image. The two application processes can then OPEN the communication line directly with:

```
OPEN (filvar, 'CSAu:, ...)
```

where:

- filvar is a Pascal file variable.
- u is a DDCMP unit number (0, 1, ...).

For example, 'CSA1:' would specify the second serial line unit listed in the CS driver prefix file.

Note

Any number of serial lines are supported, but the number is limited for each type of controller configured in the TT prefix file—up to four for DZV11, up to eight for DHV11, one for most others. The range of valid identifying unit numbers is 0 through (n–1) for n lines configured in the CS driver prefix file. Lines are numbered sequentially upward from 0 in the order they appear in the prefix file, independently of the specified TT unit numbers. Note that TT and CS unit numbers normally do not match; TTA0 is normally off limits to the CS driver because of its implicit (default) association with the standard Pascal file variables INPUT and OUTPUT.

MicroPower/Pascal Communication will place the process in a wait state until the other process issues a similar OPEN statement, at which time the two processes may begin exchanging data.

Note

The Pascal EOLN procedure is valid for TEXT data exchanged over a logical link, and the EOF procedure is valid for any link. The CLOSE and PURGE procedures are used to terminate logical links gracefully.

For examples of data transfer between two MicroPower/Pascal targets or between a MicroPower/Pascal process and a VMS image, see Chapter 11.

12.4 Request/Reply Packet Interface

The packet-level functions provided by the CS driver are listed below by symbolic and decimal function code:

Code	Function
IF$RDL (1)	Read Logical
IF$WTL (4)	Write Logical
IF$GET (7)	Get Characteristics
IF$ENA (8)	Enable Protocol
IF$DSA (9)	Disable Protocol
IF$LOK (16)	Lookup (equivalent to Enable Protocol)
IF$ENT (17)	Enter (equivalent to Enable Protocol)
IF$CLS (20)	Close (equivalent to Disable Protocol)
IF$PRG (21)	Purge (Disable Protocol variant)

Note

The MACRO–11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries; the equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the CS driver are shown below by symbolic code and bit position:

Code	Function
FM$NTR (bit 6)	Calling process (NSP) will time the line, disable keep-alive (Enable Protocol)
FM$IRP (bit 6)	Indirect reference pointers, perform gathered write (write)
FM$BSM (bit 13)	Signal binary/counting semaphore

The CS driver consists of an initialization process, which lowers its priority to become the first controller's request handler process; a timer process; and for each line, a receiver process and a transmitter process. The single request-handling process handles all serial line units specified in the CS driver prefix file. DDCMP I/O requests for any controller or line are sent (using a Pascal SEND or a MACRO–11 SEND$) to the request queue semaphore waited on by the CS static process.

The timer process runs approximately once a second to handle REP timing (the sending of "keep-alive" messages at user-defined intervals when the calling process is not timing the line), retransmissions, protocol initialization, and orderly protocol shutdown. (Shutdown can be triggered by a user request, a protocol error, or ACK timeout.)

The receiver process for each line continually reads the line's incoming byte stream, interprets DDCMP messages, and processes them according to their type. Read Logical requests (with the minimum/maximum modifier set) are sent to the TT driver on an as-needed basis.

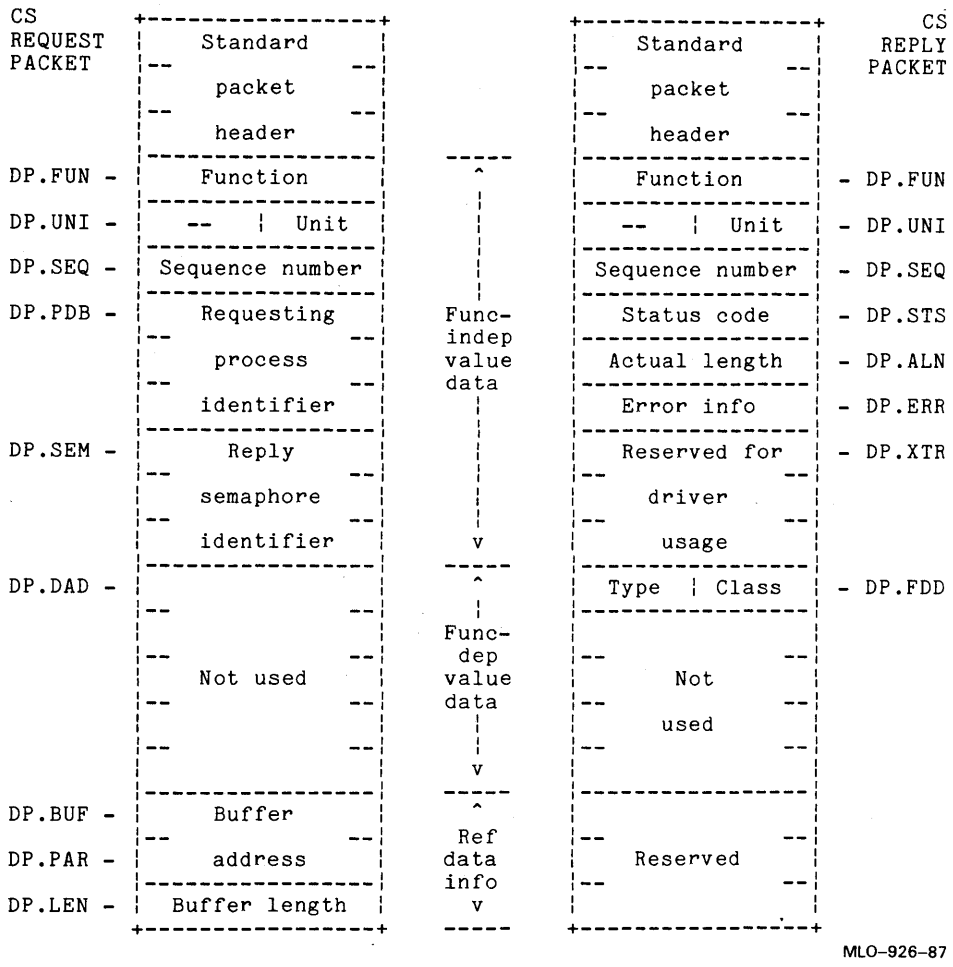
The transmitter process for each line gets messages from the line's outgoing queue, constructs the required DDCMP data messages and control messages, and executes the appropriate DDCMP sequences to send the user data. Write Logical requests are sent to the TT driver to put strings of bytes into the outgoing byte stream.

The request queue name and number of supported units for CS driver requests are:

Driver	Request Queue Name	Number of Units	Numbering
DDCMP	$CSA	1–n	0 through (n–1) in prefix file order, independently of TT unit numbers

Note that TT and CS unit numbers normally do not match; TTA0 is normally off limits to the CS driver because of its implicit (default) association with the standard Pascal file variables INPUT and OUTPUT.

The general format of the CS driver request and reply packets is shown below:



The function-independent portions of the packets shown above are described in Chapter 1, Section 1.3 (Request/Reply Packet Interface). The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The following sections describe the function-dependent portions of the request and reply packets for each type of CS driver function.

Note

The MACRO-11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO-11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

12.4.1 Enable Protocol and Disable Protocol Functions

The Enable (IF$ENA), Lookup (IF$LOK), and Enter (IF$ENT) functions each cause the timer process to initialize the protocol for a specified line. If the calling process rather than the CS driver will time the line, the calling process should issue the IF$ENA request with the FM$NTR (bit 6 in the function word) set. This disables the sending of "keep-alive" messages at user-defined intervals by the CS driver.

The Disable (IF$DSA), Close (IF$CLS), and Purge (IF$PRG) functions stop the protocol for the specified line and wait for the receiver and transmitter to shut down before returning to the user. The Purge function differs from Disable and Close only in that it causes abort (ES$ABT) status to be set for any pending I/O requests that are returned after the stop action is initiated.

The function-dependent portions of the request and reply packets are not used. You specify the unit (line) to be enabled or disabled in the unit field (offset DP.UNI) of the request packet.

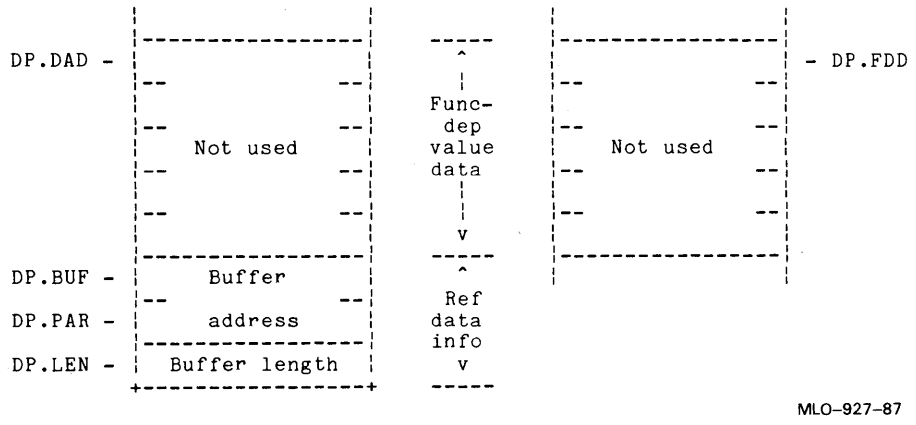
12.4.2 Read and Write Functions

Read and write functions transfer data to or from a user buffer. If the calling process is using the CS driver as a component in a higher-level protocol, the data to be received or transmitted includes control information for higher levels to interpret. Indirect reference pointers are honored for write requests.

A read request causes a user packet to be placed in the receiver queue for the specified CS serial line unit.

A write request causes the user request to be assigned a message number and placed in the transmitter queue for the specified CS serial line unit.

The function-dependent portions of the read and write request and reply packets are shown below:



MLO-927-87

For a read, the buffer-address and buffer-length fields specify the buffer that will receive data from the incoming DDCMP message stream for the specified line. The receiver process for that line interprets the incoming DDCMP messages and locates the data to be handed to the user. If more data was received from a DDCMP data message than the user requested, an

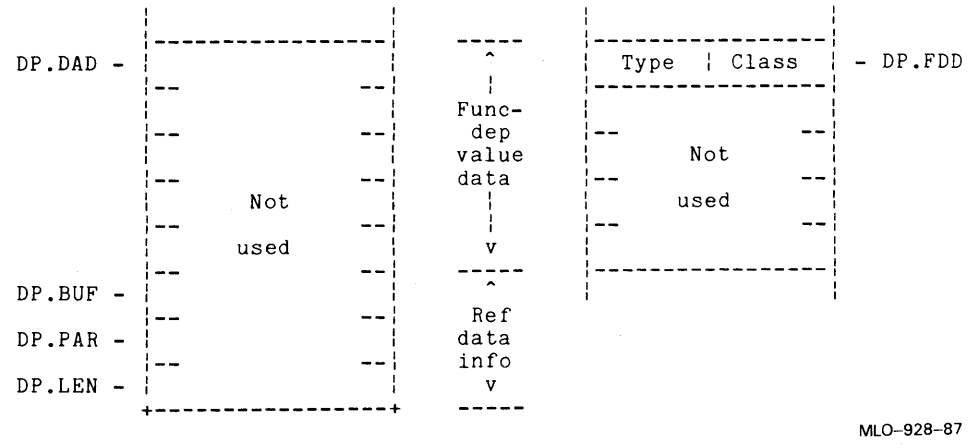
overflow (ES$OVF) error is returned, but the requested data length is copied to the buffer and the actual-length field (offset DP.ALN) of the reply filled appropriately.

For a write, the buffer-address and buffer-length fields specify the location and length of the data to be transmitted. Alternatively, if function modifier FM$IRP (bit 6 of the function word) is set, the buffer-address and buffer-length fields point to a table of 3-word data buffer specifications. Thus, the driver can do gathered writes from many buffers into one DDCMP data message. The transmitter process for the specified line frames the required DDCMP data messages and control messages and executes the appropriate DDCMP sequences to send the data.

12.4.3 Get Characteristics Function

The Get Characteristics request returns, in the reply message, codes for DDCMP "device" class and type.

The function-dependent portions of the Get Characteristics request and reply packets are shown below:



MLO-928-87

In the reply information above:

- Class is DC$PRL for protocol device class.
- Type is PR$CDM for asynchronous DDCMP communication.

12.5 Status Codes

If an error is detected during an I/O operation by a serial interface device, the asynchronous line driver, or the CS driver, the CS driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS will raise the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error is detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code (DP.STS) field.

The CS driver returns the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O aborted: user stopped protocol
ES$DAL	HARD_IO	Device already allocated: protocol already started
ES$NXU	HARD_IO	Nonexistent unit: bad unit number
ES$OVF	HARD_IO	Data buffer overflow, data truncated
ES$EOF	SOFT_IO	End of file: communication lost
ES$IFN	SOFT_IO	Illegal function code

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for I/O errors detected at higher levels or for kernel- or TT-driver-detected errors that the CS driver raises rather than passing back to the requesting process.

12.6 CS Driver Prefix File

Figure 12–1 shows the CS driver prefix module. The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

The symbols CS$IPR and CS$PPR define the initialization and request-handling software priorities for the CS driver process.

CS$TMO defines the length of time the driver will wait for a positive acknowledgment (ACK) of a message before declaring the line down; the default is 60 seconds.

CS$REP defines the interval between REP requests, which are issued to keep the line "alive" if the calling process is not timing the line. The default is 3 seconds. (REP also initiates error recovery after an ACK timeout.)

The CSDEV$ macro is invoked once for each serial line unit available to the CS driver. You specify the TT driver request semaphore name (<$TTA >) and a valid TT unit number. Remember that TT units are numbered sequentially up from 0 in the order they appear in the TT prefix file, crossing controller boundaries. CS units are also numbered sequentially up from 0 in the order they are defined, independently of the specified TT unit numbers. Note that the CS and TT unit numbers usually do not match; TTA0 is normally off limits to the CS driver

because of its implicit (default) association with the standard Pascal file variables INPUT and OUTPUT.

Terminal-oriented functions like flow control (XON/XOFF) and editing must be disabled for the CS lines; see the TT driver prefix file description in Section 3.8.

Figure 12–1: CS Driver Prefix File (CSPFX.MAC)

```
        .title  CSPFX - Serial communications driver prefix module
;+
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1984, 1986 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.
;-
       .mcall   macdf$,misdf$,cspfx$
       macdf$
       misdf$
       cspfx$

;+
; User changable symbols for CSDRV
;-

CS$IPR  ==      250.            ;Initialization priority
CS$PPR  ==      175.            ;Normal process priority
CS$TMO  ==      60.             ;Seconds without ACK before declaring line down
CS$REP  ==      3.              ;Seconds between REP requests

;+
; The definition of the serial line units available to the CSDRV process
;
; Each entry is considered an independent unit number when issuing requests
;       to the $CSA queue semaphore.  This first entry is unit 0 and so
;       forth.  Pascal programs may OPEN these units by including the unit
;       number in the device specification (e.g. OPEN(f,'CSA3:');)
;


; Each entry is in the following form:
;
;       CSDEV$   QSEM=<ssssss>,UNIT=n
;
;       ssssss   is a 6 character, upper case, blank filled device driver
;                named semaphore enclosed in angle brackets (e.g.
;                QSEM=<$TTA  >).  This parameter must be specified, there is
;                no default.
;
;       u        is the unit number if the device driver is capable of
;                supporting multiple units.  The default is 0 (zero).
;
; These units are normally those defined within the terminal driver prefix
;       file (e.g. qsem=<$TTA  >,unit=1).  When defining lines to the
;       terminal driver for use by CSDRV, you must NOT specify XON/XOFF or
;       EDITING as you would for lines connected to a terminal.  Also
;       remember that $TTA0: is normally used for the Pascal file variables
;   INPUT and OUTPUT.
;
; For example:
;
;       ttctr$ type=TT$DL, csr=176510, vector=310, hprio=4, nlines=1
;       ttlin$ ibuf=32., parm1=0, parm2=0, speed=<whatever>
;
; At this time, only lines controlled by the terminal driver are supported.
;-
```

```
;       CSDEV$  qsem=<$TTA  >,unit=0          ;$TTA0 is normally INPUT/OUTPUT
        CSDEV$  qsem=<$TTA  >,unit=1          ;Define CSA0:
;       CSDEV$  qsem=<$TTA` >,unit=2          ;Define CSA1:

        csfin$  ;Generate the data structures
        .end    ;and all done
```

Chapter 13
Communication Drivers

This chapter describes the use of the MicroPower/Pascal communication drivers, which support message-framing and error detection for point-to-point or broadcast communication with external processors. A communication driver provides the calling process with direct control of the supported communication device for the purpose of moving data. Typically, these drivers are used as "data link" drivers (ISO/DNA terminology) by the MicroPower/Pascal network service process (NSP) or by a user process that implements a communication protocol (for example, HDLC or X.25 LAPB). The calling process normally provides for guaranteed sequential message delivery (via message sequencing and retransmission on error) and, in the case of the NSP, multiplexing of "conversations" (logical links) over a physical link.

The communication drivers support the devices and protocols listed below:

Driver	Supported Devices and Protocols
QN	DEQNA Ethernet interface, Ethernet data link protocol (usable as base for DECnet)
XP	DPV11 synchronous serial line interface, bit-synchronous mode (usable as base for bit-oriented protocol, such as HDLC or LAPB)
XS	KXT11-CA/KXJ11-CA synchronous serial line interface (usable as base for bit-oriented protocol)
KK	KXT11-CA/KXJ11-CA two-port RAM, peripheral processor side of two-port RAM protocol
KX	KXT11-CA/KXJ11-CA two-port RAM, arbiter side of two-port RAM protocol

The supported devices interface an external processor to a MicroPower/Pascal target processor so that cooperating processes on the two machines can communicate.

Note

The MicroPower/Pascal definition of "communication driver" leaves out two communication-related drivers—the asynchronous serial line (TT) driver and the asynchronous DDCMP (CS) driver. Those drivers support point-to-point

communication via serial line interfaces (DLV11, DHV11, DZV11, DLARTs, KXT11-CA or KXJ11-CA multiprotocol chip).

The TT driver is not considered a communication driver, because it receives and transmits bytes rather than messages. Nevertheless, it is also usable as a data link driver within a communication protocol. (See the CS driver.) The TT driver is addressed in Chapter 3.

The CS driver is considered a protocol driver rather than a communication driver, because it does not drive a communication hardware device. Rather, it sends requests to the TT driver, which moves the data through the serial interface. Like the communication drivers, the CS driver frames messages and is used by the NSP. However, the CS driver also guarantees sequential message delivery, which communication protocol drivers nearly always do and which communication hardware drivers often do not. The CS driver is addressed in Chapter 12.

MicroPower/Pascal supports three distinct levels of communication device I/O:

- MicroPower/Pascal DECnet
- MicroPower/Pascal Communication
- Data link level (send/receive) I/O

Section 13.2 describes each level of communication device access; Table 13-1 summarizes the possible paths through the communication (and protocol) drivers.

13.1 Communication Driver Features and Capabilities

The communication drivers support read and write operations, channel enabling or disabling, and the returning of device characteristics and/or status. Indirect reference pointers are honored on write operations for the purpose of performing gathered writes.

Also, the drivers support operations that are specific to the devices or protocols they support. For driver-specific features and capabilities, see Sections 13.1.1 (Ethernet), 13.1.2 (synchronous point-to-point), and 13.1.3 (KXT11-CA/KXJ11-CA two-port RAM communication).

All communication drivers can be accessed by the NSP (which normally implies file system OTS and ACP involvement) in connection with setting up logical links and multiplexing them across physical links (virtual circuits) for task-to-task I/O or by a user process for data link level (send/receive) I/O. The KXT11-CA/KXJ11-CA two-port RAM drivers have the additional capability of being opened for Pascal file I/O without NSP involvement (for example, an OPEN of 'KXA0:').

The NSP supports the following combinations of communication (and protocol) drivers:

Drivers	Type of Communication
QN <-> DECnet	DECnet/Ethernet endnode support for communication between a MicroPower/Pascal target and another DECnet node—possibly another MicroPower/Pascal target, a VAXELN target, a VAX/VMS or RSX system, or other system running DECnet
CS <-> DECnet	DECnet/asynchronous DDCMP endnode support for communication between a MicroPower/Pascal target and another DECnet node—see Chapter 12
CS <-> CS	Asynchronous point-to-point communication between two MicroPower/Pascal targets—see Chapter 12
XP or XS <-> XP or XS	Synchronous point-to-point communication between two MicroPower/Pascal targets
KX <-> KK	Peripheral processor two-port RAM protocol between the MicroPower/Pascal arbiter and MicroPower/Pascal KXT11-CA/KXJ11-CA

13.1.1 Ethernet Communication

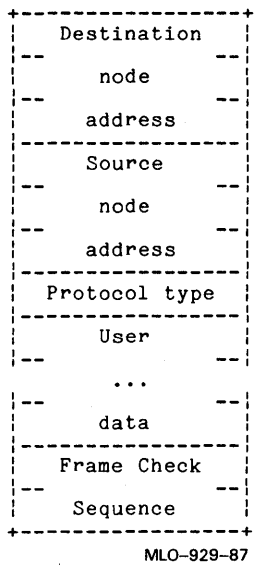
The DEQNA (QN) driver provides a standard driver interface to the Ethernet. When used with the filesystem OTS, the ACP, and the NSP, the QN driver provides an Ethernet base for MicroPower/Pascal DECnet endnode support.

When used directly via Pascal or MACRO send requests (bypassing the file system, the ACP, and the NSP), the QN driver provides probabilistic data delivery without the message retransmission or sequencing of the DECnet protocols. Direct use of the Ethernet requires a firm grasp of communication protocols. The protocol layered above the Ethernet should include a mechanism for message acknowledgment and sequencing. (See the remarks accompanying the XP/XS driver discussion, below.) Also, the Ethernet places a 1500-byte limit on the data portion of a message frame. Larger data packets should be segmented into smaller frames by the user's protocol.

Note

The QN driver plays a role, and must be present, if an application is configured to respond to network requests to trigger reloading of the target (SYSTEM debug=NO, nettrigger=YES). See Chapter 13 of the *MicroPower/Pascal-RSX/VMS System User's Guide* and Section 4.3.11 of the *MicroPower/Pascal Run-Time Services Manual* for details on DECnet/Ethernet downline loading.

Ethernet data delivery allows for individual system addressing or broadcast (multicast) addressing. Each message frame on the Ethernet consists of a destination address, a source address, a user-defined protocol number, a data portion, and a Frame Check Sequence (FCS), as follows:



MLO-929-87

The basic unit of Ethernet operation is the portal, which is a unique protocol number and a set of addresses for that protocol. The user of the QN driver issues an Enable Portal request to identify those parameters and directs the QN driver to deliver incoming messages directed to the specified addresses and protocol to the user. The QN driver is capable of supporting multiple concurrent portals, as long as each protocol/address list is unique.

The QN driver prefix file (QNPFX.MAC) allows the selection of the maximum number of portals, the number of receive buffers, and the size of each buffer. The number of and size of each buffer affect the RAM requirements of the QN driver.

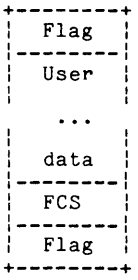
13.1.2 Synchronous Point-to-Point Communication

The XP and XS drivers provide a common user interface to the DPV11 synchronous serial line controller (XP) and the KXT11-CA/KXJ11-CA multiprotocol chip SLU2 Channel A (XS) for synchronous serial I/O. Each driver allows you to establish an elementary bit-oriented communication channel. The drivers perform the following functions of bit-oriented communication procedures:

- Synchronization (flag detection)
- Transparency (bit stuffing)
- Invalid frame detection
- Frame abortion detection
- Frame Check Sequence (FCS) checking/calculation

Either driver can be used by user-written software as a component in performing bit-oriented communication procedures, such as CCITT X.25, ISO HDLC, and others.

The XP and XS drivers send and receive frames of data and handle modem control for full-duplex channels. Basic HDLC-style framing and error detection are provided. A frame has the following general format (byte units NOT implied):



MLO-930-87

Both the leading and trailing flag bytes have the binary value 01111110 with no bit stuffing. Data sent by the driver has the FCS appended to form a frame. If any errors are detected as the driver sends the frame, the transmission is aborted and the frame resent. Frames received by the driver use the embedded FCS to verify the frame; the FCS is then removed. If the FCS checking indicates that the frame was received in error, or if the frame was aborted or invalid, the frame is discarded with no indication to the user. Otherwise, the data is returned to the user.

Because frames that are received in error are discarded, user software must be able to determine when a frame that was sent has not been received. Typically, a sequence number/timeout scheme is used for this purpose, as follows:

- A sequence number is included in the data portion of each frame.
- As the user software receives each frame, it responds by sending a frame acknowledging the receipt of the frame with that sequence number.
- After a period of time, if the originator of a frame determines that no acknowledgment for that frame has been received, it resends the frame.

13.1.3 Peripheral Processor Two-Port RAM Communication

The MicroPower/Pascal peripheral processor two-port RAM (KX and KK) drivers together implement a protocol, operating through the KXT11-CA or KXJ11-CA two-port RAM, for transfer of data between a Q-bus arbiter processor and any of up to 14 KXT11-CA or KXJ11-CA peripheral processors. The KX driver resides in the arbiter processor and supports the arbiter side of the protocol; the KK driver resides in each KXT11-CA or KXJ11-CA and supports the peripheral processor side of the protocol.

The KX and KK drivers implement the protocol via read and write commands that they issue to each other. The drivers perform as many data transfers as necessary to complete a read or write request. Each KX read or write transfers data between an arbiter buffer and the data area of a two-port RAM data channel; each KK read or write transfers data between a KXT11-CA

or KXJ11-CA buffer and a two-port RAM channel data area. The drivers operate in full duplex mode; reads and writes may go on concurrently.

The KX driver supports up to 14 KXT11-CAs or KXJ11-CAs running on the Q-bus. The KX driver communicates with each KXT11-CA/KXJ11-CA and its resident two-port RAM (KK) driver via the command and status registers of two-port RAM data channels 0 and 1 and their respective 4- and 12-byte data areas. Each data channel configured in your application—one or two per KXT11-CA or KXJ11-CA—is assigned a unit number by the KX driver for purposes of communication. Each unit is associated with a unique interrupt to the KX driver to permit fast communication. The KK driver manages the two data channels of its KXT11-CA/KXJ11-CA as separate units, numbered 0 and 1, respectively.

See Appendix B for a description of the KXT11-CA/KXJ11-CA two-port RAM protocol.

13.2 Performing Communication Device I/O

For most MicroPower/Pascal applications, communication device I/O is performed with the MicroPower/Pascal DECnet or MicroPower/Pascal communication facilities.

MicroPower/Pascal DECnet is an Ethernet (DEQNA) or asynchronous serial line-based facility that is compatible with Digital Network Architecture (DNA) products. Using MicroPower/Pascal DECnet, a MicroPower/Pascal target machine may communicate with processes in other MicroPower/Pascal targets, with processes in VAXELN targets, or with tasks in VAX/VMS, RSX, or other systems.

MicroPower/Pascal Communication allows the user to exchange data between processes on different MicroPower/Pascal target machines, using standard Pascal Input/Output statements.

Note

Transparent remote file access is not supported. One method for transferring files between systems, using task-to-task data exchange, is noted in Chapter 11.

Both the MicroPower/Pascal DECnet and Communication facilities allow sequential, error-free data delivery, while hiding the details of data exchange, such as initialization and error recovery. Both facilities can carry on many task-to-task dialogs across a single physical link. The physical link is controlled to ensure that there is no "crosstalk" between the multiplexed logical links.

The MicroPower/Pascal DECnet and Communication facilities have the following components:

- The NSP coordinates the flow of data between two processes. The NSP conforms to the DNA specifications for the Session Control layer, the End Communication (Network Services) layer, and the Routing layer of DECnet. See Chapter 11 for details.
- The CS driver monitors the data transfer between two MicroPower/Pascal target machines and performs the appropriate recovery algorithms to correct transmission errors. The CS driver conforms to the DNA specification for the Digital Data Communications Message Protocol (DDCMP) and the DECnet Data Link layer. (CS is a protocol driver and does not control hardware directly; it uses the TT driver to perform lower-level data link functions. See Chapter 12 for details.)
- The communication drivers (QN, XP, XS, KX, and KK) control communication device hardware, performing such data link functions as message framing and error detection.

Both MicroPower/Pascal DECnet and Communication treat task-to-task communication as a normal input/output device. Programs written to communicate with other tasks use standard Pascal I/O statements. The data structure controlling access and interpretation of the data exchanged by the programs is the file variable. The OPEN statement establishes logical links between both active and passive tasks; the HISTORY and I/O specification arguments to the OPEN procedure create active and passive links. GET, PUT, READ, and WRITE statements transfer data over the logical link. EOLN is valid on text files over logical links, and EOF is valid on any link. CLOSE and PURGE are used to terminate logical links gracefully. Chapter 9 of the *MicroPower/Pascal Language Guide* describes the exact syntax of the Pascal I/O statements.

Note

The following applies to KXT11–CA/KXJ11–CA two-port RAM communication only. When multiplexing of logical links over a KXT11–CA/KXJ11–CA two-port RAM data channel is not necessary, you can eliminate the NSP from your application and open the data channel by specifying "KXA0:" (for example) in the OPEN statement. Nonmultiplexing is a special case of MicroPower/Pascal Communication and applies only to communication and protocol drivers that allow direct opens—KX, KK, and CS.

For each participating MicroPower/Pascal application, in addition to invoking the Pascal I/O procedures, you must:

1. Edit the DEVICES configuration macro to reflect the communication controller interrupt vector addresses; if the QN driver is required, edit the PROCESSOR macro in the system configuration file to specify a clock argument (for line timing) and edit the DEVICES configuration macro to reflect the clock interrupt vector addresses
2. Edit the communication driver prefix file(s) to reflect:
   - Number of controllers
   - [For each controller:] Controller identifier (A, B, ...), CSR address, interrupt vector address, and number of controller units (portals for QN, two-port RAM data channels for KX)
   - Hardware interrupt priority
   - Other interface characteristics, such as the number and size of DEQNA receive buffers, DEQNA unit (portal) numbers, or XP/XS station address
   - Driver initialization and request-handling process priorities
3. [For logical link OPEN:] Edit the NSP prefix file to define the communication devices available to the NSP process. Non-DECnet devices precede a DECnet device, and only one DECnet device may be specified. Specify TYPE parameter ETHER for DEQNA. For DPV11, KXT11–CA/KXJ11–CA synchronous, or KXT11–CA/KXJ11–CA two-port RAM, specify type UPOWER and supply an address (parameters AREA and ADDRESS).
4. [For OPEN:] Edit the ACP prefix file to indicate whether NSP open support is required; the default is inclusion of NSP open support. Also, check the ACP pool size; 180 bytes are required per NSP open.

5. Build into each participating MicroPower/Pascal application the following I/O system components:
   - Communication driver process
   - [For logical link OPEN:] The NSP
   - [For OPEN:] The ACP
   - Pascal OTS routines for file service—built in automatically by MPBUILD for programs that invoke Pascal I/O procedures—plus any I/O support routines you opt to include (see kit files GETSET.PAS and GSINC.PAS)

For more information on setting up your application software for communication device I/O, see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*, Section 13.7 of this manual, and the material on building system processes in the MicroPower/Pascal system user's guide for your host system.

When a module that contains Pascal I/O procedure invocations is built into your application, Pascal OTS routines for file service are linked to the module. The OTS file routines perform all Pascal operations on logical-link or device files, including opening, input, and output. In particular, they perform the necessary low-level processing of high-level operations such as OPEN and WRITE. Thus, the basic mechanisms of MicroPower/Pascal I/O—the sending of request packets to driver or NSP or ACP queue semaphores, the dispatching of interrupts, and the signaling of reply semaphores—are concealed from the Pascal user.

As alternatives to using the MicroPower/Pascal DECnet or Communication facilities for communication device I/O, you can:

- Issue your own Pascal or MACRO–11 packet-level requests to the ACP and an NSP logical-link server (or KXT11–CA/KXJ11–CA two-port RAM driver if no NSP), bypassing the OTS file routines (lower-level file system access)
- Issue your own Pascal or MACRO–11 packet-level requests to the drivers, bypassing the OTS file routines, the ACP, and the NSP (low-level nonfile access); alternatively, existing applications that require them can invoke Pascal routines that support nonfile access (see Section 13.7)

Table 13–1 summarizes the possible paths through the communication (and protocol) drivers. For reference, the three levels of communication represented are MicroPower/Pascal DECnet, MicroPower/Pascal Communication, and data link level (send/receive) I/O.

Table 13–1: Communication I/O Paths and Interfaces

Type of Communication	I/O System Components			Interface	
	ACP/NSP	**Prot**	**Comm/TT**	**Pascal**	**MACRO-11**
DECnet/Ethernet	ACP/NSP	–	QN	OPEN link	SEND$ to ACP (or NSP using ACP format)
Ethernet (data link level)	–	–	QN	SEND to driver	SEND$ to driver
DECnet/DDCMP or DDCMP Communication	ACP/NSP	CS	TT	OPEN link	SEND$ to ACP (or NSP using ACP format)
DDCMP non-mux Communication	ACP	CS	TT	OPEN comm.	SEND$ to ACP line
DDCMP (data link level)	–	CS	TT	SEND to driver	SEND$ to driver
Synchronous serial Communication	ACP/NSP	–	XP/XS	OPEN link	SEND$ to ACP (or NSP using ACP format)
Synchronous serial (data link level)	–	–	XP/XS	SEND to driver	SEND$ to driver
KXT11–CA/ KXJ11–CA TPR Communication	ACP/NSP	–	KX/KK	OPEN link	SEND$ to ACP (or NSP using ACP format)
KXT11–CA/ KXJ11–CA TPR nonmux Communication	ACP	–	KX/KK	OPEN device	SEND$ to ACP
KXT11–CA/ KXJ11–CA TPR (data link level)	–	–	KX/KK	Call support	SEND$ to driver routines or SEND to driver

The following sections describe the Pascal I/O statement interfaces to the communication drivers, the lower-level request/reply packet interface, the status codes that can be returned to users of any interface, and the communication driver prefix files.

13.3 Pascal I/O Procedure Interface

To perform standard Pascal I/O to a communication device, you issue an OPEN statement that associates a file variable with a logical link—or, optionally for the KXT11-CA/KXJ11-CA two-port RAM, a device channel. The file variable controls access and interpretation of data as it is exchanged, via the communication device, with a cooperating program on a different machine. No restriction is placed on the type or contents of the exchanged data, as long as all is consistent with the method of defining files in Pascal. Programs may use TEXT, FILE of CHAR, FILE of INTEGER, and so forth when defining the content of the messages passed between them. Nor must both programs use the same definition.

To accomplish the dialog between tasks, one task must take the role of the initiator, or active task, while the other defines itself as the target, or passive task. The passive task must define a name that identifies the task to an active task; the active task specifies the name of the passive task when initiating the connection. Task names can be up to 16 characters long. When initiating the connection, the active task must specify the machine containing the passive task it is attempting to locate. MicroPower/Pascal DECnet will dynamically determine the location of each named machine. For MicroPower/Pascal Communication, each data channel is associated with a static name that identifies the machine to which it is connected.

Once the connection between the two tasks is established, the dialog is bidirectional; that is, either task may WRITE/PUT data to it or READ/GET data from it, and eventually close it, as if it were a file residing on a local device. The synchronization of data direction is the responsibility of the two programs.

For example, a passive task can issue an OPEN statement of the form:

```
OPEN (fvar, 'SY$NET:"TASK=taskname"', HISTORY:=NEW);
```

This specifies that the program is establishing itself as a passive task with the name "taskname". The program will remain in a wait state until an active task initiates a connection.

The active task issues an OPEN statement of the form:

```
OPEN (fvar, 'node::"TASK=taskname"', HISTORY:=OLD);
```

This specifies that the program is initiating a connection to a passive task named "taskname" located on node "node" that is waiting to accept a connection. The syntax of the OPEN is compatible with VAX/VMS Pascal syntax for task-to-task communication.

A passive task that is already engaged in a dialog with another task is ineligible to accept another connection. (A file variable describes exactly one task-to-task dialog.) If you want multiple active tasks to initiate dialogs with a common passive task, the passive task must, at the completion of the OPEN statement, spawn another process that issues the identical OPEN statement. The new task will then be available for subsequent connections.

The preceding discussion of the OPEN statement applies to MicroPower/Pascal DECnet and to MicroPower/Pascal Communication in the case where several "conversations" are occurring on the data channel. However, for KXT11-CA/KXJ11-CA two-port RAM communication, if the data channel is to be used by a single process in one target talking to a single process in another target, you can avoid the overhead of the NSP process simply by not installing it into the application image. The two application processes can then OPEN the communication line directly with:

```
OPEN (filvar, 'ddcu:', ...)
```

where:

- filvar is a Pascal file variable.
- dd is the driver identifier (KX for arbiter side, KK for peripheral processor side).
- c is a KXT11-CA/KXJ11-CA identifier (A through N for KX, A for KK).
- u is a KXT11-CA/KXJ11-CA unit number (0, 1).

For example, 'KXA0:' would specify the first unit (0) of the first KXT11-CA/KXJ11-CA (A) listed in the KX driver prefix file.

Note

Up to two units are supported for each KXT11-CA or KXJ11-CA. KX units are numbered 0 and 1 in the order that their CSR and vector values are specified in the KX driver prefix file. KK unit numbers are 0 for data channel 0 and 1 for data channel 1.

MicroPower/Pascal Communication will place the process in a wait state until the other process issues a similar OPEN statement, at which time the two processes may begin exchanging data.

Note

The Pascal EOLN procedure is valid for TEXT data exchanged over a logical link, and the EOF procedure is valid for any link. The CLOSE and PURGE procedures are used to terminate logical links gracefully.

For examples of data transfer between two MicroPower/Pascal targets or between a MicroPower/Pascal process and a VMS image, see Chapter 11.

13.4 Request/Reply Packet Interface

The packet-level functions provided by the communication drivers are listed below by symbolic and decimal function code:

Code	Function
IF$RDP (0)	Read Physical (equivalent to Read Logical)
IF$RDL (1)	Read Logical
IF$WTP (3)	Write Physical (equivalent to Write Logical)
IF$WTL (4)	Write Logical
IF$GET (7)	Get Characteristics
IF$ENA (8)	Enable
IF$DSA (9)	Disable
IF$STP (10)	Stop Requests (DPV11, KXT11-CA/KXJ11-CA synchronous)
IF$SMD (11)	Set Modem Semaphore (DPV11, KXT11-CA/KXJ11-CA synchronous)

The DEQNA, DPV11, and KXT11-CA/KXJ11-CA synchronous drivers cannot be opened directly—that is, without NSP involvement—because of the handshakes they perform. If a request is received for an Open (IF$LOK or IF$ENT), the driver returns an unsupported function

code (ES$UFN). This causes the OTS to raise the exception, provided the OTS/ACP issued the Open request and the user's OPEN statement did not specify a status return.

The KXT11-CA/KXJ11-CA two-port RAM drivers allow direct opens (for example, an OPEN of 'KXA0'). If a request is received for an Open (IF$LOK or IF$ENT), an illegal function status code (ES$IFN), which the ACP (Open) or OTS (Close/Purge) interprets as indicating that no device-dependent processing was required for that operation.

Note

The MACRO-11 symbols used in this section are defined by the DRVDF$ macro, which resides in the COMU and COMM kernel macro libraries. The equivalent Pascal symbols are defined in the IOPKTS.PAS include file.

The function modifiers recognized by the communication drivers are shown below by symbolic code and bit position:

Code	Function
FM$IRP (bit 6)	Indirect reference pointers, perform gathered write (Write Logical)
FM$ANY (bit 6)	Enable promiscuous mode (QN Enable)
FM$RAD (bit 7)	Recognize address (XP or XS Enable)
FM$KRR (bit 6)	Kill read requests (XP or XS Stop)
FM$KWR (bit 7)	Kill write requests (XP or XS Stop)
FM$BSM (bit 13)	Signal binary/counting semaphore

The QN, XP, XS, and KX drivers each consist of an initialization process, which lowers its priority to become the first controller's request handler process, plus an additional request handler process for each configured controller. (If a nonzero timer value is specified in the QN driver prefix file, the QN driver starts up an internal timer process as well.) I/O requests for a controller are sent (by means of a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by that controller's request handler process.

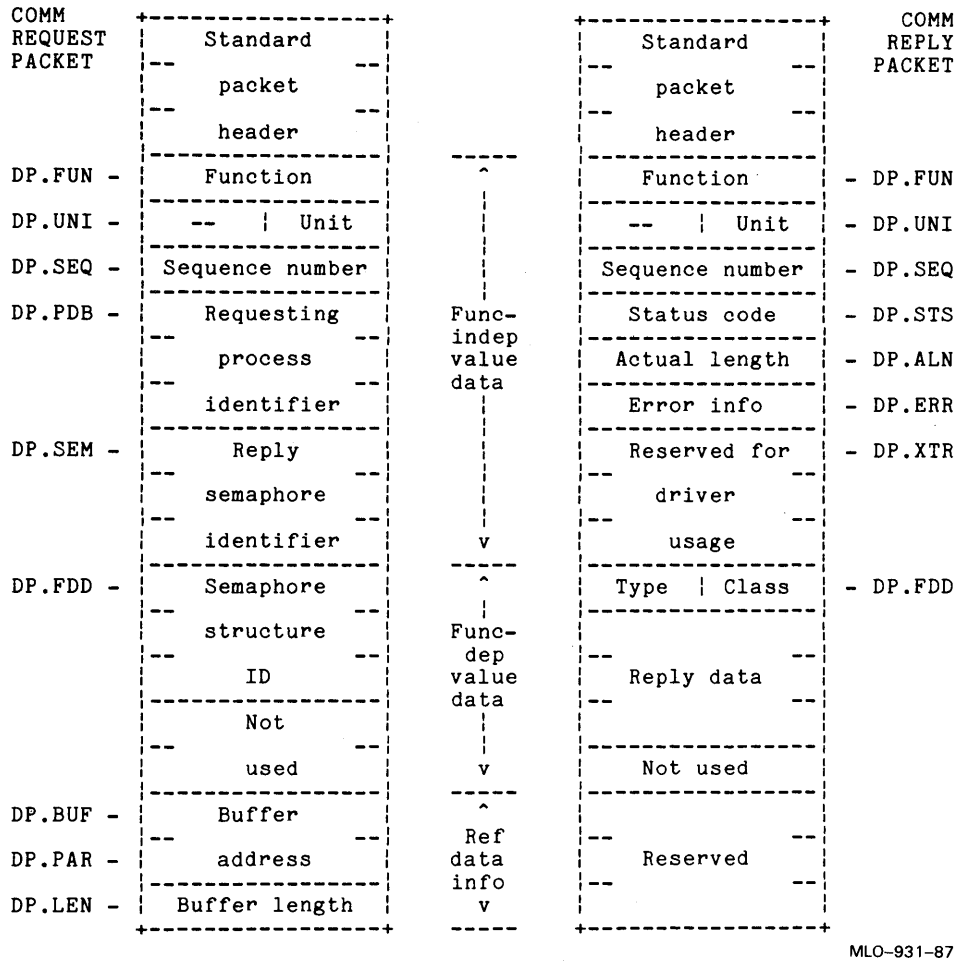
The KK driver is a single (static) process, beginning as an initialization process and then lowering its priority to the running level specified in the KK prefix file. I/O requests are sent (using a Pascal SEND or a MACRO-11 SEND$) to the request queue semaphore waited on by the driver process.

The request queue names and number of supported units for communication driver requests are shown below:

Driver	Request Queue Name	Number of Units	Numbering
DEQNA	$QNc	1–4 (portals)	In prefix file
DPV11	$XPc	1	0
KXT11–CA or KXJ11–CA synchronous	$XSc	1	0
KXT11–CA or KXJ11–CA TPR	$KXc	1–2	0 and 1 in prefix file order
KXT11–CA or KXJ11–CA TPR	$KKA	1–2	0 for channel 0 and 1 for channel 1

The letter c in a queue name represents a controller designation (A, B, ..., as specified in a driver prefix file). The number of units configured for each controller must be specified in a prefix file.

The general format of the communication device request and reply packets is shown below:



MLO-931-87

The function-independent portions of the packets shown above are described in Chapter 1, Section 1.3 (Request/Reply Packet Interface). The valid function and function-modifier codes for the function (DP.FUN) field and the valid unit numbers for the unit (DP.UNI) field are listed at the beginning of this section.

The function-dependent portions of the request and reply packets are described in the sections that follow for each type of communication driver function.

Note

The MACRO–11 field names shown above do not represent offsets into the user's send or reply buffers; they are offset symbols used by MACRO–11 drivers to reference packets. For example, DP.FUN is a 6-byte offset from the packet header.

13.4.1 DEQNA (QN) Functions

13.4.1.1 QN Enable Portal

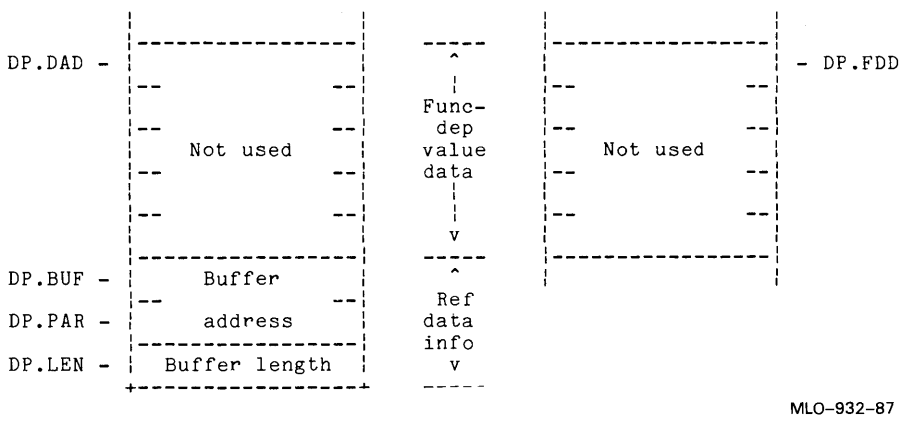
The Enable Portal (IF$ENA) function initializes a portal and returns a prefix-file-assigned portal number in the unit field of the reply packet. Once a portal has been enabled, read and write requests can be issued, using the returned portal (unit) number.

The portal is the basic unit of QN operation. It consists of a unique user-defined protocol number and a set of 48-bit Ethernet addresses for that protocol. Protocol numbers and addresses are kept in an active portal list; the addresses are also placed in an address recognition table.

An Enable request specifies the protocol number and the addresses to be enabled for a portal.

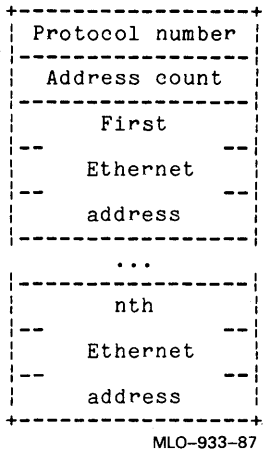
If the modifier bit FM$ANY (bit 6 in the function word) is set, promiscuous mode is enabled, causing the DEQNA driver to return all Ethernet messages without address/protocol checks. No other portals may be enabled if promiscuous mode is enabled. The function-dependent portions of the QN Enable request and reply packets are not used for an enable promiscuous mode request.

For nonpromiscuous Enables, the function-dependent portions of the request and reply packets are as follows:



MLO-932-87

The buffer-address and buffer-length fields give the location and length of an address/protocol buffer, constructed as follows:



MLO-933-87

The maximum number of addresses per portal is currently defined as four. Additionally, the DEQNA restricts the total number of unique addresses for all portals (that is, the maximum number of address recognition table entries) to 14. If an Enable request specifies more than four addresses or causes the recognition table to overflow, an error is returned and the portal is not enabled.

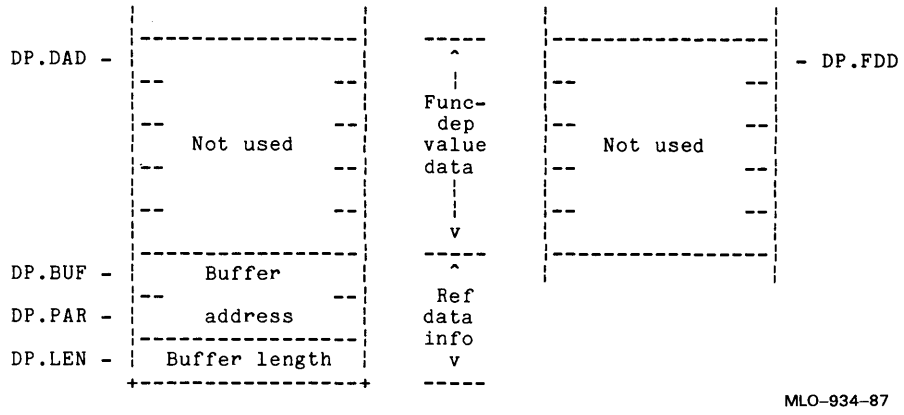
13.4.1.2 QN Read and Write

A DEQNA read (IF$RDL) request for a correctly enabled portal causes a user packet to be placed in the read queue for the specified portal (unit). Incoming messages from the Ethernet are checked against the addresses and protocol number currently enabled for the portal. If a protocol/address match is found, the data is copied from the receive buffer to a user-specified buffer. If no match was found, the message is discarded.

If promiscuous mode is enabled (see the Enable function), read data is copied from the receive buffer to the user buffer with no protocol/address checking.

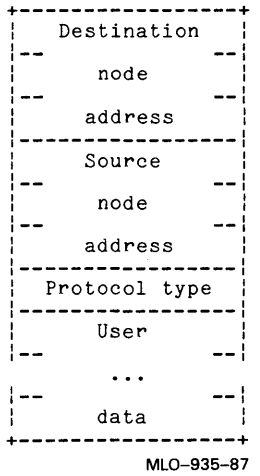
A write (IF$WTL) request for a correctly enabled portal causes a user-specified buffer to be transmitted via the DEQNA interface.

The function-dependent portions of the QN read or write request and reply packets are shown below:



MLO-934-87

For a read, the buffer-address and buffer-length fields specify the buffer that will receive the data. After a successful read, the user's buffer contains an Ethernet header (destination, source, and type) and the Ethernet data, as follows:



MLO-935-87

The Frame Check Sequence (FCS) is not returned.

No Ethernet message is returned to more than one read request. If more data was received from the Ethernet message than the user requested, an overflow error (ES$OVF) is returned, but the requested data length is copied to the buffer, and the actual-length field (offset DP.ALN) filled appropriately.

For a write, the data to be written must include the standard Ethernet header, as shown above. The buffer-address and buffer-length fields specify the location and length of the data buffer. Alternatively, if write function modifier FM$IRP (bit 6 of the function word) is set, the buffer-address and buffer-length fields point to a table of 3-word data buffer specifications. Thus, the driver can do gathered writes from many buffers into one Ethernet packet.

If the source-node-address field for a write is all zeros, the driver automatically fills in the board address; otherwise, the specified address is used.

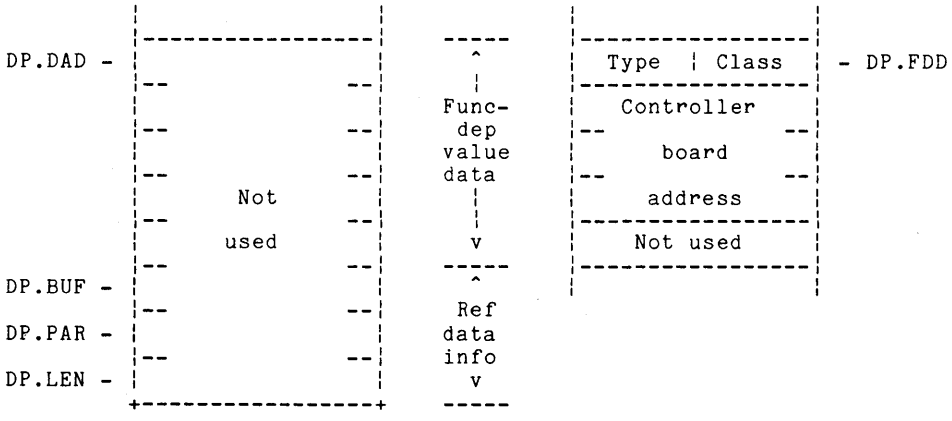
If the message to be written is too short, the driver zero-fills the message to the minimum length. The minimum length of an Ethernet message is 60 (decimal) bytes, including the destination, source, and protocol fields (14 bytes).

If the DEQNA hardware collision detection/retry logic indicates a failure to transmit the data, an "unsafe" error (ES$UNS) is returned.

13.4.1.3 QN Get Characteristics

The Get Characteristics request returns codes for device class and type, plus the 48-bit Ethernet node address of the controller in use.

The function-dependent portions of the QN Get Characteristics request and reply packets are shown below:



MLO-936-87

In the reply information above:

- Class is DC$COM for communication device class.
- Type is CM$ETH for Ethernet device type.

13.4.1.4 QN Disable Portal

The Disable Portal (IF$DSA) function disables a portal and removes the portal's addresses from the address recognition table.

The function-dependent portions of the QN Disable request and reply packets are not used. You specify the portal to be disabled in the unit field (offset DP.UNI) of the function-independent portion of the request packet.

13.4.2 DPV11 and KXT11-CA/KXJ11-CA Synchronous Communication (XP and XS) Functions

13.4.2.1 XP or XS Enable and Disable

The XP or XS Enable function turns a line on and readies it for communication. It also enables interrupts on transitions of modem control signals. (See Section 13.4.2.5.)

If the function modifier FM$RAD (bit 7 in the function word) is set, automatic recognition of the secondary station address is also enabled. Only messages prefixed with the correct secondary address reach the user.

The XP or XS Disable function turns a line off and disconnects it from any communication. Any outstanding I/O requests to the line are returned with abort (ES$ABT) status.

The function-dependent portions of the XP or XS Enable and Disable request and reply packets are not used.

13.4.2.2 XP or XS Read and Write

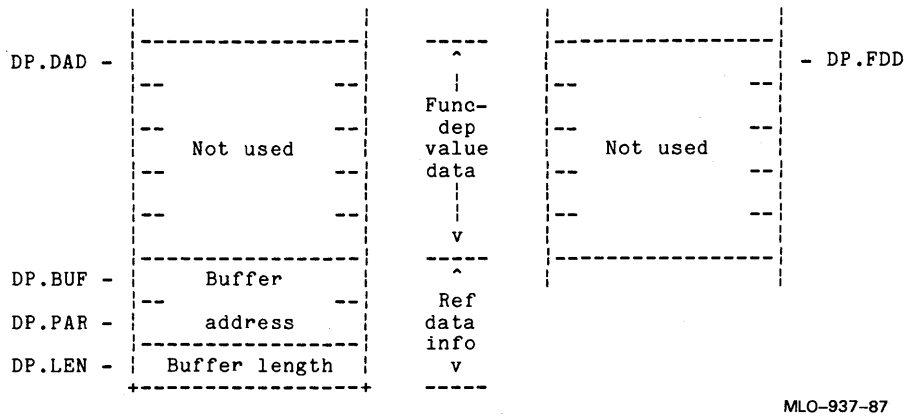
The XP or XS read function causes the next correctly received frame of data to be passed from the device to a user-specified buffer.

The XP or XS write function causes a user-supplied buffer of data to be transmitted as a single frame to the device. If a transmission underflow occurs, the frame is retransmitted.

Because the XP and XS drivers perform bit-oriented communication functions—flag detection, bit stuffing, invalid frame detection, frame abortion detection, and FCS checking/calculation—any frame returned by a read request is a valid frame with a correct FCS and with all bit stuffing removed. Correspondingly, frames transmitted by the write request have appropriate bit stuffing and FCS added by the driver.

The indirect reference pointer modifier FM$IRP (bit 6 of the function word) is honored for write requests. This allows the driver to perform gathered writes from many buffers into one frame.

The function-dependent portions of the XP or XS read and write request and reply packets are shown below:



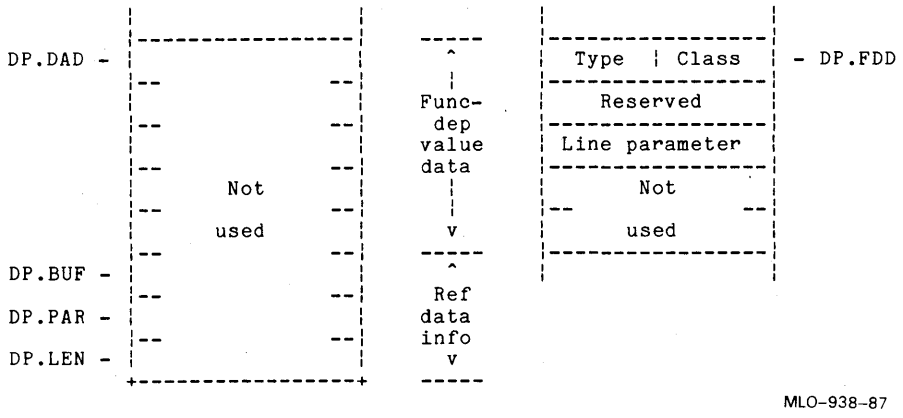
For a read, the buffer-address and buffer-length fields specify the buffer into which a frame will be read. The frame returned in the buffer does not include the FCS.

For a write, the buffer-address and buffer-length fields specify the location and length of the data to be written. Alternatively, if write function modifier FM$IRP is set, the buffer-address and buffer-length fields point to a table of 3-word data buffer specifications.

13.4.2.3 XP or XS Get Characteristics

The Get Characteristics request returns codes for device class and type, plus bit settings for the Ring, Carrier Detect, Clear to Send, and Data Set Ready modem controls.

The function-dependent portions of the XP or XS Get Characteristics request and reply packets are shown below:

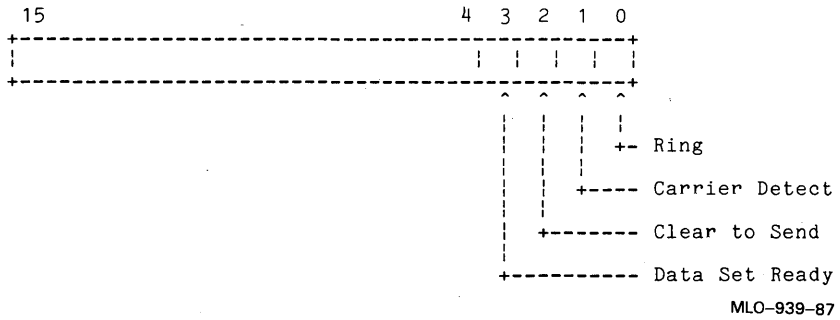


MLO-938-87

In the preceding reply information:

- Class is DC$COM for communication device class.
- Type is CM$DPV for DPV11 device type, or CM$XSK for KXT11-CA/KXJ11-CA synchronous communication device type.

The format of the line parameter is shown below:



MLO-939-87

Bits 0 (Ring) through 3 (Data Set Ready) are modem control bits. Proceeding from right to left in the format above:

- Bit 0, if set, indicates a Ring, informing the target processor that an incoming call signal is being received by the modem.
- Bit 1, if set, indicates Carrier Detect, informing the target processor that the data channel signal is OK and the receiver is ready.
- Bit 2, if set, indicates Clear to Send, informing the target processor that the modem is ready to transmit data.
- Bit 3, if set, indicates Data Set Ready, informing the target processor that the modem is in data mode and ready to operate.

13.4.2.4 XP or XS Stop

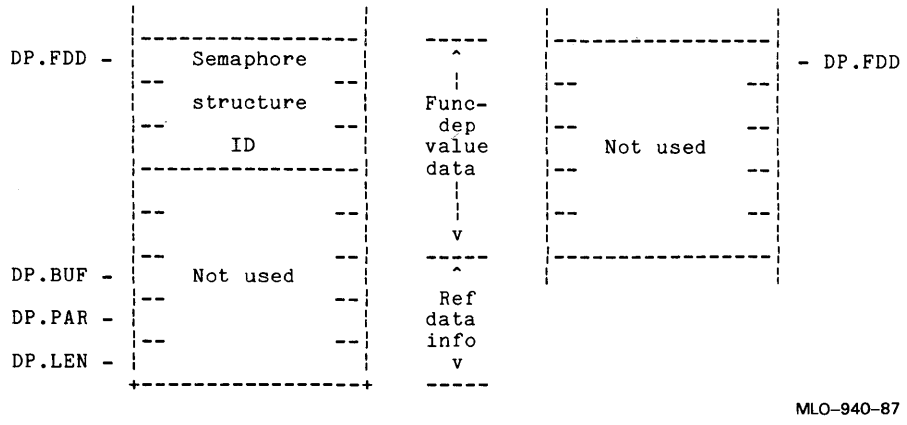
The XP or XS Stop function causes all pending reads and/or writes to be returned with abort (ES$ABT) status.

The function-dependent portions of the XP or XS Stop request and reply packets are not used. You select the type of requests to be stopped by setting the read requests bit (FM$KRR), the write requests bit (FM$KWR), or both, in the function word of the request packet.

13.4.2.5 XP or XS Set Modem Semaphore

The Set Modem Semaphore (IF$SMD) function specifies the binary or counting semaphore to be signaled upon each modem interrupt. Modem interrupts are generated when a change in modem status occurs on a specified line. Modem interrupts are enabled for a line when the line is enabled. (See Section 13.4.2.1.)

The function-dependent portions of the Set Modem Semaphore request and reply packets are shown below:



MLO-940-87

The binary or counting semaphore specified at offset DP.FDD is signaled whenever a modem control interrupt occurs on the line.

The calling program is responsible for issuing a Get Characteristics request to determine the current status on each signal.

13.4.3 KXT11-CA/KXJ11-CA Two-Port RAM (KX and KK) Functions

13.4.3.1 KX or KK Read and Write

Read and write operations transfer data between a buffer in an arbiter process and a buffer in a KXT11-CA/KXJ11-CA process, via a two-port RAM data channel. All transfers are initiated by the KX driver placing a read or write command in the command register for a user-specified data channel; however, for the transfer to occur, there must be a matching request queued for that channel at the KK driver end. In other words, KX writes must be matched by KK reads, and KX reads must be matched by KK writes. To help synchronize arbiter and KXT11-CA/KXJ11-CA transfer requests, the KK driver can interrupt the KX driver when data is available or when data has been requested at the KXT11-CA/KXJ11-CA end.

The KX and KK drivers operate in full duplex mode; reads and writes may go on concurrently.

The indirect reference pointer modifier FM$IRP (bit 6 of the function word) is honored for write requests by both drivers. This allows the drivers to perform gathered writes from many buffers in one request.

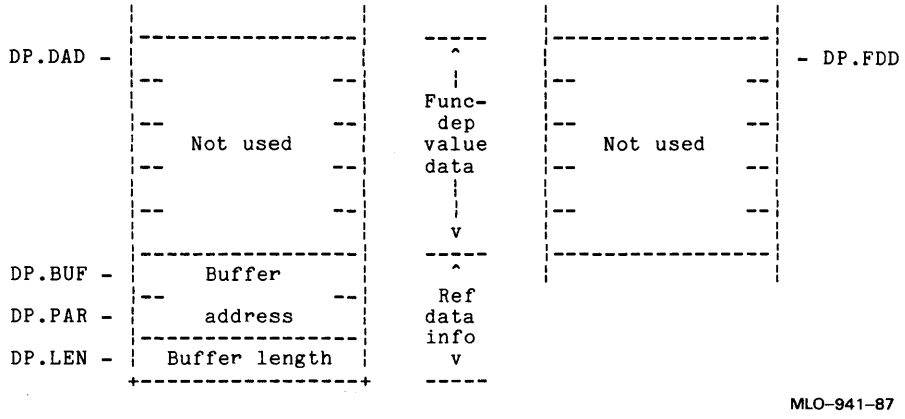
The read functions (IF$RDP and IF$RDL) instruct the KX driver to get data from a specified unit and place it in a user-specified data area. If necessary, the request is held until the KXT11-CA/KXJ11-CA with the specified unit number becomes ready.

The write functions (IF$WTP and IF$WTL) instruct the KX driver to send data to a specified unit from a user-specified data area. If necessary, the request is held until the KXT11-CA/KXJ11-CA with the specified unit number becomes ready.

The read functions (IF$RDP and IF$RDL) instruct the KK driver to get data from the two-port RAM on the Q-bus. The immediate action caused by the request is the setting of a "data requested" bit and the interrupting of the Q-bus, if bus interruption is enabled. The read request is queued and completes when the arbiter transfers data across the Q-bus to satisfy the request. The data is placed in a user-specified data area.

The write functions (IF$WTP and IF$WTL) instruct the KK driver to move data from a user-specified area across the Q-bus to the arbiter. The arbiter must issue a corresponding read request before the request can complete. The immediate action caused by the request is the setting of a "data available" bit and the interrupting of the arbiter, if interruption is enabled. The request is queued until the arbiter issues a read request to take the data.

The function-dependent portions of the read and write request and reply packets are shown below:



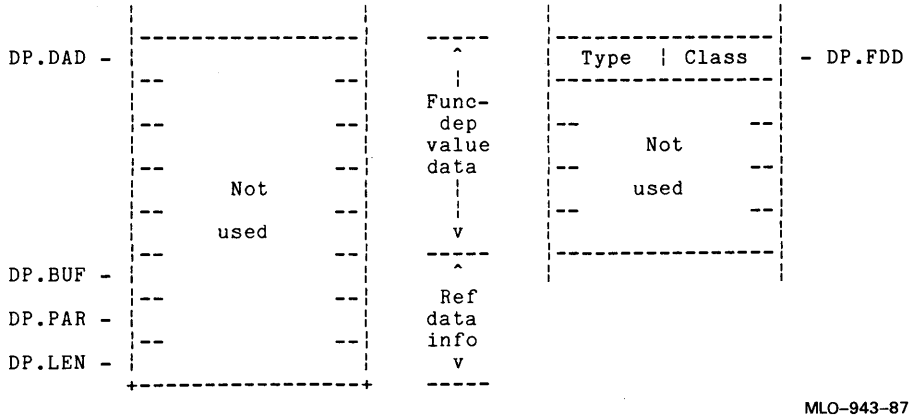
For a read, the buffer-address and buffer-length fields specify the buffer into which data will be read.

For a write, the buffer-address and buffer-length fields specify the location and length of the data to be written. Alternatively, if write function modifier FM$IRP is set, the buffer-address and buffer-length fields point to a table of 3-word data buffer specifications.

13.4.3.2 KX or KK Get Characteristics

The Get Characteristics function returns bit settings that indicate the two-port RAM device class and the two-port RAM driver type in the function-dependent portion of the reply message.

The function-dependent portions of the Get Characteristics request and reply packets are shown below:



In the information above:

- Class is DC$COM for communication device class.
- Type is CM$KXK for arbiter two-port RAM driver, or CM$KKK for peripheral processor two-port RAM driver.

13.4.3.3 KX or KK Enable and Disable

The KX or KK Enable and Disable requests return with normal status without performing any channel-related operations.

13.5 Status Codes

If an error is detected by a communication device or driver during an I/O operation, the driver returns an exception code in the status-code (DP.STS) field of the reply message. If you are performing I/O with Pascal I/O statements—that is, not with send/receive statements or Pascal support routine calls—the Pascal OTS raises the corresponding exception (unless the operation was an OPEN for which a STATUS return was specified). If no error was detected during the I/O operation, a value of ES$NOR (0) is returned in the status-code field.

The communication drivers return the following exception codes:

Code	Type	Description
ES$ABT	HARD_IO	I/O request canceled by user or aborted by remote node
ES$DAL	HARD_IO	Device already allocated: line already enabled (XP, XS)
ES$IVD	HARD_IO	Invalid data: too many addresses for portal or for address recognition table (QN)
ES$IVM	HARD_IO	Invalid mode: promiscuous mode already enabled or enabled with portals active (QN)
ES$NXU	HARD_IO	Nonexistent unit (QN, KX, KK)
ES$OVF	HARD_IO	Data overflow: received data overflows data buffer
ES$UNS	HARD_IO	Unsafe volume: line not enabled (XP, XS); interface not on (KX); failure to transmit detected by DEQNA (QN)
ES$DCF	SOFT_IO	Device full: no portals available (QN)
ES$IFN	SOFT_IO	Illegal function code; also used internally by KX and KK drivers to signal ACP or OTS that no device-dependent processing of an Open, Close, or Purge was required
ES$IVL	SOFT_IO	Invalid data length specified (QN)
ES$NRF	SOFT_IO	No reference data present
ES$UFN	SOFT_IO	Unsupported function; file open (for example, OPEN of 'QNA0:') attempted (QN, XP, XS)

Exception codes are defined in the ESCODE.PAS include file (included by EXC.PAS) for Pascal users and by the EXMSK$ macro in the COMU/COMM macro libraries for MACRO-11 users.

Note

Not listed above are exception codes for I/O errors detected at higher levels or for kernel-detected errors that the communication drivers raise rather than passing back to the requesting process.

13.6 Communication Driver Prefix Files

Figures 13–1 through 13–5 show the communication driver prefix modules. The following sections describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

13.6.1 QN Prefix File

The symbols QN$IPR, QN$PPR, and QN$HPR define the initialization and request-handling software priorities for the Ethernet driver process and the hardware interrupt priority for the DEQNA controller(s).

The DRVCF$ macro defines the number of DEQNA controllers on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request-queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced. It gives the controller name, the maximum number of portals, CSR and vector addresses, the number of receive buffers (in the range 3 to 12), the size of each receive buffer (in the decimal range 256 to 1514), a timer interval value (10 to 60 seconds in msecs., or 0), and portal unit numbers.

Note

The driver's internal timer process checks at user-specified intervals to determine if the DEQNA board is operational. This is done to protect against the tendency of some versions of the DEQNA board to lock up. If no messages have been received from the board in the specified interval, the timer process assumes that the board is locked and resets the board (requeuing any transmit that was pending).

The timer interval is specified via the "timer" value in the prefix file. DIGITAL recommends that the interval remain set at 20 seconds (in msecs.) for applications that use the DEQNA board in a DECnet/Ethernet environment.

Figure 13–1: DEQNA Driver Prefix File (QNPFX.MAC)

```
    .TITLE   QNPFX - DEQNA Ethernet driver prefix file
;
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;
    .mcall   drvcf$,ctrcf$

; Define the hardware and software priorities associated with the QNA.

QN$IPR   ==   250.       ; Initialization priority
QN$PPR   ==   175.       ; Process priority
QN$HPR   ==   4          ; Hardware priority

;+
; 3 <= numbufs <= 12.
; 256. <= size <= 1514.
;  10 seconds <= timer <= 60. seconds
;-
    numbuf = 4.
    bufsz  = 512.
    numprt = 3              ; number of portals available
    timer  = <20.*1000.>    ; timer in msecs (20 seconds)

drvcf$ dname=QN, nctrl=1
ctrcf$
cname=A,nunits=numprt,csrvec=<174440,400,numbuf,bufsz,timer>,units=<0:2>

    .END
```

13.6.2 XP and XS Prefix Files

The symbols XxIPR, XxPPR, and Xx$HPR define the initialization and request-handling software priorities for the XP or XS driver process and the hardware interrupt priority for the DPV11 or KXT11–CA/KXJ11–CA multiprotocol chip serial line channel A controller(s).

The DRVCF$ macro defines the number of controllers on the target to be supported by the driver. The dname field specifies the first two characters of the corresponding request-queue semaphore name.

The CTRCF$ macro is invoked once for each controller to be serviced. It gives the controller name, number of units (1), CSR and vector addresses, and station address. The unit number is 0.

Figure 13–2: DPV11 Driver Prefix File (XPPFX.MAC)

```
        .TITLE   XPPFX   - DPV11 Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
        .mcall   drvcf$, ctrcf$

XP$IPR   ==   250.       ; Process initialization priority
XP$PPR   ==   175.       ; Process running priority
XP$HPR   ==   6          ; Hardware priority

        drvcf$   dname=XP, nctrl=1
        ctrcf$   cname=A, nunits=1, csrvec=<160010,500,123>

        .end
```

Figure 13–3: KXT11–CA/KXJ11–CA Synchronous Serial Driver Prefix File (XSPFX.MAC)

```
    .TITLE   XSPFX   - KXT-11 Bit Synchronous Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
    .mcall   drvcf$, ctrcf$

XS$IPR   ==   250.       ; Process initialization priority
XS$PPR   ==   175.       ; Process running priority
XS$HPR   ==   4          ; Hardware priority

    drvcf$   dname=XS, nctrl=1
    ctrcf$   cname=A, nunits=1, csrvec=<175700,140,123>

    .end
```

13.6.3 KX and KK Prefix Files

The KX prefix module invokes the DRVCF$ and CTRCF$ macros and assigns hardware and driver process priorities on the arbiter side.

The DRVCF$ macro specifies the device name (KX) and number of controllers. Each KXT11–CA/KXJ11–CA on the arbiter is considered to be one controller. Use the nctrl argument of the DRVCF$ macro to specify the number of KXT11–CA/KXJ11–CA boards that are plugged into the Q-bus.

Use one CTRCF$ macro to configure each KXT11–CA or KXJ11–CA; you should have parameter in the DRVCF$ macro. The nunits parameter can have a value of 1 or 2. Unit numbers for each CTRCF$ macro are allocated, starting at 0, in the order the CSR/vector pairs are given. The CSR/vector pairs can be specified in any order. The CTRCF$ macro ignores any value given for the units parameter; specify units= <0> .

Note that the specified interrupt vectors must also be specified in the system configuration file, using the DEVICES macro.

Table 13–2 shows the two-port RAM data channel addresses associated with each KXT11–CA/KXJ11–CA identification (ID) switch position. Note that for each KXT11–CA or KXJ11–CA in your system, you must select an identification switch position that is unique in the system and either a high or low base address range.

Table 13–2: Two-Port RAM Data Channel Addresses

KXT11-CA/ KXJ11-CA ID Switch Position	High-Range Channel Address (Jumper: In=KXT, Out=KXJ)		Low-Range Channel Address (Jumper: Out=KXT, In=KXJ)	
	Channel 0	Channel 1	Channel 0	Channel 1
0	STAND—ALONE MODE			
1	STAND—ALONE MODE			
2	17762110	17762120	17760110	17760120
3	17762150	17762160	17760150	17760160
4	17762210	17762220	17760210	17760220
5	17762250	17762260	17760250	17760260
6	17762310	17762320	17760310	17760320
7	17762350	17762360	17760350	17760360
8	17777410	17777420	17775410	17775420
9	17777450	17777460	17775450	17775460
10	17777510	17777520	17775510	17775520
11	17777550	17777560	17775550	17775560
12	17777610	17777620	17775610	17775620
13	17777650	17777660	17775650	17775660
14	17777710	17777720	17775710	17775720
15	17777750	17777760	17775750	17775760

Table 13–3 shows the default ID, CSR address, and interrupt vector values that are supplied in the KX driver prefix file (as distributed on the MicroPower/Pascal kit). These values assume low base address ranges for all 14 KXT11–CAs or KXJ11–CAs (2 through 15).

Table 13–3: KX Prefix File Defaults

KXT11-CA/ KXJ11-CA ID Switch Position	Default Controller ID ($KXx)	Default Unit 0 CSR	Default Unit 0 Vector	Default Unit 1 CSR	Default Unit 1 Vector
0	STAND-ALONE MODE				
1	STAND-ALONE MODE				
2	A	160110	500	160120	504
3	B	160150	510	160160	514

Table 13–3 (Cont.): KX Prefix File Defaults

KXT11-CA/ KXJ11-CA ID Switch Position	Default Controller ID ($KXx)	Default Unit 0 CSR	Default Unit 0 Vector	Default Unit 1 CSR	Default Unit 1 Vector
4	C	160210	520	160220	524
5	D	160250	530	160260	534
6	E	160310	540	160320	544
7	F	160350	550	160360	554
8	G	175410	560	175420	564
9	H	175450	570	175460	574
10	I	175510	600	175520	604
11	J	175550	610	175560	614
12	K	175610	620	175620	624
13	L	175650	630	175660	634
14	M	175710	640	175720	644
15	N	175750	650	175760	654

There are no controller or unit parameters to modify in the KK prefix module for the KXT11–CA/KXJ11–CA side. These parameters are always the same for each KXT11–CA or KXJ11–CA—Controller A and units 0 and 1. The CSR for unit 0 is 175010, and the vector is 120; the CSR for unit 1 is 175020, and the vector is 124. The module defines hardware and driver process priorities and references the global $KK, which extracts the KK driver from the device driver library during the application build.

Figure 13–4: KXT11–CA/KXJ11–CA Two-Port RAM Driver Prefix File (KXPFX.MAC)

```
   .title   KXPFX   - KXT11--CA/KXJ11--CA Two port memory device driver
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 By Digital Equipment Corporation.
; All rights reserved.
;-

   .SBTTL   Edit History
;+
;
; Module name:  KXPFX.MAC
;
; System: MicroPower/Pascal Prefix files
;
;-

   .mcall   drvcf$
   .mcall   ctrcf$

KX$PPR   ==   175.      ; Process priority
KX$HPR   ==   4         ; Hardware priority
KX$IPR   ==   250.      ; Process initialization priority

    drvcf$   dname=KX,nctrl=1
    ctrcf$   cname=A,nunits=2.,csrvec=<160110,500,160120,504>,units=<0>
;   ctrcf$   cname=B,nunits=2.,csrvec=<160150,510,160160,514>,units=<0>
;   ctrcf$   cname=C,nunits=2.,csrvec=<160210,520,160220,524>,units=<0>
;   ctrcf$   cname=D,nunits=2.,csrvec=<160250,530,160260,534>,units=<0>
;   ctrcf$   cname=E,nunits=2.,csrvec=<160310,540,160320,544>,units=<0>
;   ctrcf$   cname=F,nunits=2.,csrvec=<160350,550,160360,554>,units=<0>
;   ctrcf$   cname=G,nunits=2.,csrvec=<175410,560,175420,564>,units=<0>
;   ctrcf$   cname=H,nunits=2.,csrvec=<175450,570,175460,574>,units=<0>
;   ctrcf$   cname=I,nunits=2.,csrvec=<175510,600,175520,604>,units=<0>
;   ctrcf$   cname=J,nunits=2.,csrvec=<175550,610,175560,614>,units=<0>
;   ctrcf$   cname=K,nunits=2.,csrvec=<175610,620,175620,624>,units=<0>
;   ctrcf$   cname=L,nunits=2.,csrvec=<175650,630,175660,634>,units=<0>
;   ctrcf$   cname=M,nunits=2.,csrvec=<175710,640,175720,644>,units=<0>
;   ctrcf$   cname=N,nunits=2.,csrvec=<175750,650,175760,654>,units=<0>

   .end
```

Figure 13–5: KXT11–CA/KXJ11–CA Two-Port RAM Driver Prefix File (KKPFX.MAC)

```
.TITLE   KKPFX - KXT11--CA/KXJ11--CA TWO PORT RAM DEVICE DRIVER PREFIX
MODULE
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1984, 1986 By Digital Equipment Corporation.
;      All rights reserved.
;-

   .GLOBL   $KK

KK$HPR   == 5          ; Hardware priority
KK$IPR   == 250.       ; Initialization priority
KK$PPR   == 175.       ; Process priority

   .END
```

13.7 Peripheral Processor Communication Support Routines

The following Pascal routines provide a nonfile-oriented interface to the KXT11–CA/KXJ11–CA two-port RAM data channels:

- KX_READ_DATA function
- KX_WRITE_DATA function
- KK_READ_DATA function
- KK_WRITE_DATA function

Note

These routines are provided primarily for existing applications (developed with Version 1 of MicroPower/Pascal) that require them. They perform all packet-level driver functions except Get Characteristics (IF$GET). A non-file-oriented Get Characteristics function is provided in the distribution kit file GETSET.PAS.

The following sections describe the Pascal functions for non-file-oriented two-port RAM I/O. Each function allocates an I/O packet, fills it with information based on the function parameters, and sends it to the KX or KK driver.

If a reply queue semaphore is specified in the function call, the function returns immediately after sending the driver request. When the operation is complete, the driver sends a standard device driver reply via the specified semaphore. (The driver reply is described in Section 13.4.) The completion status and the actual length returned in the reply packet must be processed by a routine that is waiting on the semaphore.

If no reply semaphore is provided, the function waits for the two-port RAM driver reply before returning to the caller.

The following files on the MicroPower/Pascal distribution kit are required for using the functions:

File	Description
KXRWD.PAS	KX function source module
KKRWD.PAS	KK function source module
KXINC.PAS	KX function include file
KKINC.PAS	KK function include file
IOPKTS.PAS	Pascal I/O include file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include files must be included in the program at compile time.

The following data structures, referenced further below, define the KX and KK unit numbers for this interface:

```
TYPE
 $KX_unit = 0..1;
 $KK_unit = 0..1;
```

See Section 13.4 for more information about KXT11-CA/KXJ11-CA unit numbers.

13.7.1 KX_READ_DATA

The KX_READ_DATA function transfers data from a KXT11-CA/KXJ11-CA buffer to an arbiter buffer and returns a completion-status value of type UNSIGNED. See Section 13.5 for a list of completion-status values.

The syntax for calling this function is as follows:

```
KX_READ_DATA ( buffer,length,ret_length,controller,unit,reply,seq_num )
```

Parameter	Type	Description
VAR buffer	UNIVERSAL	Data buffer
length	UNSIGNED	Buffer length
VAR ret_length	UNSIGNED	Variable that returns number of bytes actually transferred—not returned if reply parameter provided
controller	CHAR	Optional controller (KXT11-CA/KXJ11-CA) designation letter; default is 'A'
unit	$KX_unit	Optional unit number specifying the unit to send requests to; default is 0

Parameter	Type	Description
reply	STRUCTURE_DESC_PTR	Optional pointer to initialized reply queue semaphore descriptor; default is NIL, which causes function to create necessary semaphore for user, then deletes it at end of function call
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet)

The buffer and length parameters specify the location and length of the buffer into which data will be read.

If no reply parameter is provided, the function sets the parameter ret_length to the number of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the KX driver reply packet.

13.7.2 KX_WRITE_DATA

The KX_WRITE_DATA function transfers data from an arbiter buffer to a KXT11-CA or KXJ11-CA buffer and returns a completion-status value of type UNSIGNED. See Section 13.5 for a list of completion-status values.

The syntax for calling this function is as follows:

```
KX_WRITE_DATA ( buffer,length,ret_length,controller,unit,reply,seq_num )
```

Parameter	Type	Description
VAR buffer	UNIVERSAL	Data buffer
length	UNSIGNED	Buffer length
VAR ret_length	UNSIGNED	Variable that returns number of bytes actually transferred—not returned if reply parameter is provided
controller	CHAR	Optional controller (KXT11-CA/KXJ11-CA) designation letter; default is 'A'
unit	$KX_unit	Optional unit number specifying the unit to send the data to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to initialized reply queue semaphore descriptor; default is NIL, which causes function to create necessary semaphore for user, then delete it at end of function call
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet)

The buffer and length parameters specify the location and length of the data to be sent through the dual-port registers.

If no reply parameter is provided, the function sets the parameter ret_length to the number of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the KX driver reply packet.

13.7.3 KK_READ_DATA

The KK_READ_DATA function transfers data from the arbiter to a KXT11-CA/KXJ11-CA buffer and returns a completion-status value of type UNSIGNED. See Section 13.5 for a list of completion-status values.

The syntax for calling this function is as follows:

```
KK_READ_DATA ( buffer,length,ret_length,unit,reply,seq_num )
```

Parameter	Type	Description
VAR buffer	UNIVERSAL	Data buffer
length	UNSIGNED	Buffer length
VAR ret_length	UNSIGNED	Variable that returns number of bytes actually transferred—not returned if reply parameter provided
unit	$KK_unit	Optional unit number specifying unit the request is to be sent to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to initialized reply queue semaphore descriptor; default is NIL
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet)

If no reply parameter is provided, the function sets the parameter ret_length to the number of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the KK driver reply packet.

13.7.4 KK_WRITE_DATA

The KK_WRITE_DATA function transfers data from a KXT11-CA or KXJ11-CA buffer to the arbiter and returns a completion-status value of type UNSIGNED. See Section 13.5 for a list of completion-status values.

The syntax for calling this function is as follows:

```
KK_WRITE_DATA ( buffer,length,ret_length,unit,reply,seq_num )
```

Parameter	Type	Description
VAR buffer	UNIVERSAL	Data buffer
length	UNSIGNED	Buffer length

Parameter	Type	Description
VAR ret_length	UNSIGNED	Variable that returns number of bytes actually transferred—not returned if reply parameter provided
unit	$KK_unit	Optional unit number specifying unit that data is to be sent to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to initialized reply queue semaphore descriptor; default is NIL
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in driver reply packet; default is 0 (0 is returned in reply packet)

If no reply parameter is provided, the function sets the parameter ret_length to the number of bytes transferred by the operation. Otherwise, the count of bytes transferred is returned in the actual-length field of the KK driver reply packet.

Chapter 14
Guide to Writing a Device Driver

This chapter explains how to write custom device drivers—device drivers that are not supplied in your MicroPower/Pascal distribution kit—for use in MicroPower/Pascal applications. DIGITAL intends to maintain the existing interface between device drivers and the kernel. However, because of the intimate relationship between the kernel and device drivers, some unavoidable changes may occur as new features are added to new versions of the MicroPower/Pascal product. Thus, device drivers written for the current version of MicroPower/Pascal may require modification for use with later versions.

14.1 Device Driver Overview

A device driver is a set of processes, routines, and tables that process I/O requests for a hardware device or device controller. In general, device drivers are restricted to device-specific aspects of I/O processing; device-independent I/O processing common to other drivers or system services is performed by other system components.

Device drivers process I/O requests by performing many or all of the following functions:

- Defining the peripheral device for the system
- Preparing the device hardware and/or its controller for operation at system start-up
- Performing device-dependent I/O preprocessing
- Translating programmed requests for I/O operations into device-specific hardware commands
- Activating—starting or enabling—the device
- Responding to any interrupt requests generated by the device hardware; a device driver can also poll devices
- Responding to requests to stop (abort) the I/O operation
- Reporting device errors to the requesting process
- Returning completion status—successful completion or error—to the process that requested the I/O operation

A process requests I/O service from a particular device by sending a request packet to the device driver's request queue. The communication between the requesting process and the device driver, including request and reply packet formats, is described in detail in Chapter 1. Interrupt dispatching and interrupt service routines are described in detail in Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual.* You must become thoroughly familiar with the technical material presented in those chapters before attempting to write a device driver.

The global symbol names shown in this chapter have 2-letter device identifiers, which also appear as part of the module name. (For example, the 2-letter device identifiers shown as xx are replaced with DY in the RX02 device driver.) DIGITAL reserves the range ZA to ZZ for customer use. Thus, in order to avoid conflicts, use identifiers in the range ZA to ZZ only for device drivers that you write.

A dollar sign ($) in the module name identifies the symbol as an address, constant, or macro. A dollar sign as the first character in the name indicates that the symbol is an address. If the dollar sign is the third character in the name, the symbol is a constant. If the dollar sign is the last character in the name, the symbol is a macro.

When adding a custom device driver to your application, you will generally write or edit three separate source modules:

- The driver prefix module (xxPFX.MAC or xxPFX.PAS)
- The driver impure-area definition macro (xxISZ$) or program (if Pascal)
- The driver proper (xxDRV.MAC or xxDRV.PAS)

Device drivers designed and written by DIGITAL include the three source modules listed above. The driver prefix module and impure-area definition macro make it easier to modify driver operations to conform to a particular hardware configuration. However, if the configuration of your hardware is not likely to change, you can write a device driver without using either the driver prefix module or the driver impure-area macro.

When designing and writing device drivers, carefully consider the conventions presented in the remaining sections of this chapter. DIGITAL discourages the use of kernel interfaces in device driver designs other than those described in this chapter.

When writing source modules in MACRO–11, follow the sample coding standard contained in Appendix E of the *PDP–11 MACRO–11 Language Reference Manual.* Also see the sample MACRO–11 device driver listed in Appendix D as a guide for designing and writing your custom device driver.

The remaining sections in this chapter contain specific guidelines for writing each module.

14.2 Device Driver Prefix Module

The device driver prefix module statically allocates the impure area required for the device driver and defines certain device-specific parameters, such as:

- Priority values for the driver initialization process, request-handling process, and device hardware interrupt priority
- Number of hardware device controllers that the driver must support
- CSR address for each controller
- Interrupt vector address for each controller
- Number of units and unit numbers supported on each controller

The device-specific information is available to the driver in the form of tables of constants and text.

One device driver prefix module is generally required for each device driver. The module name has the format xxPFX.MAC (or xxPFX.PAS), where xx identifies the device driver supported by the module. For example, DYPFX.MAC is the device driver prefix module for DYDRV.MAC, the RX02 device driver.

A driver prefix module generally contains two or more macro invocations, which you must edit in order to specify the hardware that is to be supported by the device driver. The specific macros invoked in the prefix module are the driver configuration macro (DRVCF$) and the controller configuration macro (CTRCF$), which reside in the COMM and COMU kernel macro libraries. (Some drivers and driver-related system processes have special requirements and invoke neither the DRVCF$ nor the CTRCF$ macros; for example, see the TT, CS, and NSP prefix files.)

14.2.1 Priority Assignments

The driver prefix module defines global symbols for process and hardware priority, as follows:

Parameter	Definition
xx$PPR	Process priority for controller process
xx$HPR	Hardware interrupt priority
xx$IPR	Initialization process priority

DIGITAL recommends an initialization process priority (xx$IPR) of 250 and a controller process priority (xx$PPR) of 175. Use of those recommended (default) priorities helps avoid unnecessary context switching in most cases. You can, however, specify higher or lower priorities as the application requires.

Note

In the current version of MicroPower/Pascal, all initialization procedures in Pascal processes execute at software process priority 248. The initialization priorities are currently followed only by DIGITAL-supplied device drivers written in MACRO–11. This may change in a future release.

Hardware device interrupt requests at levels 4 to 7 are supported only by LSI-11/23, SBC-11/21, and LSI-11/73 microcomputers. LSI-11 and LSI-11/2 microcomputers support interrupt requests only at level 4. (That is, only bit 7 in the PSW controls interrupts. When the bit is set, interrupts are disabled; when the bit is cleared, interrupts are enabled.)

Device hardware interrupt priority relative to two or more devices at the same interrupt request level is determined by the relative electrical position of each device along the LSI-11 bus. The device electrically closest to the microcomputer module receives the highest interrupt priority; similarly, the device farthest from the microcomputer receives the lowest priority at a particular interrupt request level.

User processes requesting I/O operations should have priorities less than or equal to those of drivers unless the application requirements for a high priority process dictate otherwise.

14.2.2 DRVCF$ Macro

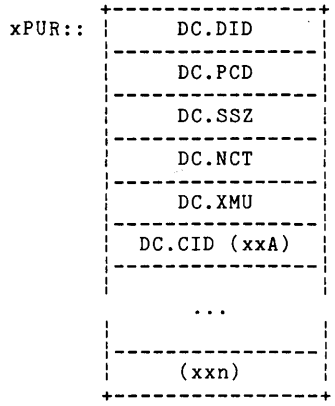
DRVCF$, the first macro to be invoked, is the driver configuration macro and specifies the driver prefix and the number of controllers to be supported by the driver. DRVCF$ defines global symbols used for configuration. The DRVCF$ macro and its parameters are as follows:

```
DRVCF$  dname=xx,nctrl=n
```

Parameter	Definition
xx	2-letter device identifier
n	Integer specifying the number of controllers that the device driver must support

When executed, DRVCF$ invokes the device driver impure-area definition macro (xxISZ$), which is device-specific. DRVCF$ directly or indirectly defines the following global symbols:

Parameter	Definition
xx$ISZ	Number of bytes needed for the fixed part of the impure area by a controller process
xx$SSZ	Number of bytes needed for stack space per controller process, excluding guard words
xx$USZ	Number of additional bytes needed for each unit supported by the driver
xx$MXU	Maximum number of units that can be supported by a single controller
$xxIMP	Address of the driver impure area
xx$NUM	Number of controllers to be supported in this configuration
$xxPUR	Address of the driver configuration data; DRVCF$ produces the data structure shown below for the device driver initialization process; names prefixed with DC. are offsets from $xxPUR



MLO-944-87

Name	Definition
DC.DID	2-letter device identifier (xx)
DC.PCD	Pointer to the device driver process-creation data ($xxPCD); label $xxPCD:: immediately precedes a CRPC$P macro call in the driver-proper source (see Section 14.4.4.3); the reference to $xxPCD generated by DRVCF$ causes the driver to be brought in from the driver object library
DC.SSZ	Process stack size, in bytes (xx$SSZ)
DC.NCT	Number of controllers supported by a single controller (nctrl)
DC.XMU	Maximum number of units supported by a controller (xx$MXU)
DC.CID	Pointer to the controller A initialization data; additional data words follow, one for each controller (B, C, ... n), as required

The DRVCF$ macro must always be invoked before the controller configuration macro CTRCF$.

14.2.3 CTRCF$ Macro

Separate invocations of the controller configuration macro (CTRCF$) are required for each controller supported. Each CTRCF$ macro specifies:

- The driver prefix
- The controller code (A, B, C, and so forth)
- The addresses of the controller's CSRs and interrupt vectors
- The numbers of the specific units supported on the controller
- The hardware type and extra parameters, if used by the driver being configured

The CTRCF$ macro and its parameters are as follows:

```
CTRCF$ cname,nunits,<csr1,vec1[,csr2,vec2...,csrn,vecn]>,units,
       <type1,xprm1[,type2,xprm2...,typen,xprmn]>
```

Parameter	Definition
cname	1-letter controller identifier
nunits	Integer specifying the number of units that the controller must support
csr1,vec1	CSR address for the first unit and address of the controller's first interrupt vector
csr2,vec2, csrn,vecn	Parameter pairs for additional units (2 to n)

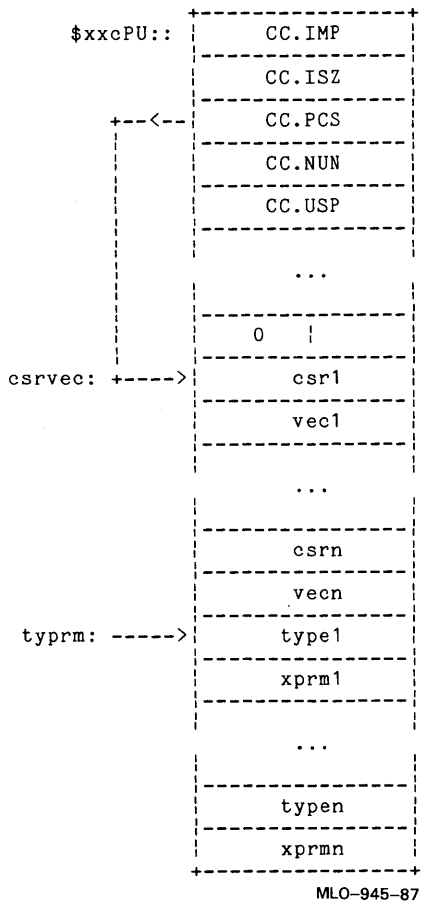
units	Angle-bracketed list of integers, specifying the unit numbers of the units supported on the controller. You can specify unit numbers in one of two ways. You can enumerate the unit numbers explicitly, separating them with commas:
	`nunits=8.,units=<1,2,3,4,5,6,7,8>`
	You can also use a colon (:) to indicate a range of unit numbers:
	`nunits=8.,units=<0,2,7:12>`
	Note
	The units parameter is ignored by some drivers, which may instead assume a single unit number of 0. The individual driver chapters describe the unit numbering for each distributed driver.

type1	Asynchronous serial line type (TU58 driver) or ISR buffer size (analog-to-digital converter driver) associated with the first CSR; not used by other standard drivers but available for user-written drivers
xprm1	Baud rate (TU58 driver); not used by other standard drivers but available as extra parameter for the first CSR for user-written drivers
typen	Serial line type (TU58 driver) or other parameter for the nth CSR
xprmn	Baud rate (TU58 driver) or extra parameter for the nth CSR **Note** The type-and-parameter pair is omitted for most drivers.

The CTRCF$ macro defines the symbols listed below. The xx characters in the symbols represent a 2-letter device identifier. The letter c in the symbols represents a letter that identifies an individual controller. (For example, $DYACS is the correct symbol for the CSR address for the RX02, controller A.)

Parameter	Definition
$xxcG1	Address of the low stack guard word for controller c
$xxcG2	Address of the high stack guard word for controller c
$xxcIM	Address of the impure area for controller c
$xxcPU	Address of the pure-code configuration data for controller c
$xxcCS	CSR address for controller c
$xxcVE	Interrupt vector address for controller c

CTRCF$ produces the data structure shown below for the device driver initialization process. Names prefixed with CC. are offsets from $xxcPU.



Name	Definition
CC.IMP	Address of the impure area for controller c ($xxcIM)
CC.ISZ	Number of bytes in the impure area for controller c (impsiz)
CC.PCS	Pointer to the CSR/vector-pair list (csrvec)
CC.NUN	Number of units supported by controller c (nunits)
CC.USP	Specification of the unit numbers supported by the controller; a list of integers separated by commas or colons and terminated by a zero byte; a pair of integers separated by a colon represents a range of unit numbers—for example, 2:4 specifies units 2, 3, and 4
csrvec	A series of word pairs containing the CSR address and vector address for each unit
typrm	A series of word pairs containing extra parameters for each unit—zero if not used by the driver being configured; among the standard drivers, only the TU58 and analog-to-digital converter drivers use typrm

CTRCF$ generates the controller impure area, as follows:

```
$xxcIM::
                .blkb   xx$ISZ  ; Impure area
                .even
$xxcG1::        .blkw           ; Low stack guard word
                .blkb   xx$SSZ  ; Stack space
$xxcG2::        .blkw           ; High stack guard word
```

The symbols xx$ISZ and xx$SSZ are defined by the impure-area definition macro, xxISZ$, which is invoked by the DRVCF$ macro. See Sections 14.2.2 and 14.3 for more information.

14.2.4 Sample Driver Prefix Module (DYPFX.MAC)

The following is an example of a driver prefix module.

```
        .NLIST
        .ENABL   LC
        .LIST

   .TITLE   DYPFX   - RX02 Prefix File
;+
; This software is furnished under a license and may be used or copied
; only in accordance with the terms of such license.
;
; Copyright (c) 1982, 1986 by Digital Equipment Corporation.
; All rights reserved.
;-
   .mcall   drvcf$, ctrcf$

DY$IPR    ==      250.      ; Process initialization priority
DY$PPR    ==      175.      ; Process priority
DY$HPR    ==      5         ; RX02 hardware priority

          drvcf$   dname=DY,nctrl=1
          ctrcf$   cname=A,nunits=2.,csrvec=<177170,264>,units=<0:1>
;         ctrcf$   cname=B,nunits=2.,csrvec=<177200,270>,units=<0,1>
.end
```

14.3 Device Driver Impure-Area Definition Macro (xxISZ$)

The device driver impure-area definition macro defines configuration parameters required by the device driver prefix module. When adding a custom device driver to the system, build a macro library with the device driver impure-area definition macro in it, for use during assembly of the driver and prefix files. (The impure-area definition macros for the standard device drivers reside in the COMU and COMM macro libraries.) The macro should include the following parameters:

Parameter	Definition
xx$ISZ	Number of bytes needed for the fixed part of the impure area by a controller process
xx$SSZ	Number of bytes needed for stack space per controller process **Note** Process stack size requirements depend on whether the system is mapped or unmapped. The symbol $MINST defines the minimum process stack size—the number of bytes on the process stack required by the kernel for its own use. The value of $MINST is larger for unmapped systems than for mapped systems. Processes can make optimum use of memory by defining process stack requirements in terms of $MINST (defined in the MISDF$ macro). For example: `mxx$SSZ = $MINST+n` In the example above, n is the number of bytes on the process stack needed by the process itself.
xx$USZ	Number of additional bytes needed for each unit supported by the driver
xx$MXU	Maximum number of units that can be supported by a single controller

You obtain the values for the symbols xx$ISZ and xx$USZ from the assembly listing of the driver proper. When these values change through program modification, edit the xxISZ$ macro to reflect the changes.

The DY driver impure-area definition macro, DYISZ$, defines configuration parameters as follows:

```
.MACRO          DYISZ$ range=nogbl
.if             NDF $MINST
                .mcall MISDF$
                misdf$
.endc
.MACRO          DRVR.. X
DY$USZ          ='X 0
DY$SSZ          ='X $MINST+110
DY$ISZ          ='X 112
DY$MXU          ='X 2
.ENDM           DRVR..
                .dsabl LC
.IF             IDN <range>,<GLOBAL>
                drvr.. <=>
.IFF
                drvr.. <>
.ENDC
                .enabl LC
.ENDM           DYISZ$
```

14.4 Device Driver Proper

The device driver proper is the device driver source module. (The device driver prefix module and the impure-area definition macro previously described provide configuration information required by the driver proper.) The following paragraphs describe the composition of the device driver source module.

By convention, a device driver source has a file name of the form xxDRV.MAC. In general, device drivers supplied by DIGITAL in the MicroPower/Pascal distribution kit conform to MicroPower/Pascal and MACRO–11 coding conventions. You can use the RX02 device driver MACRO–11 source file, DYDRV.MAC, as an example when you write device drivers in MACRO–11; a listing of DYDRV.MAC is in Appendix D.

When writing device drivers in Pascal, refer to the YADRV.PAS source code for the DRV11 device driver; YADRV.PAS and its prefix file, YAPFX.PAS, are included on the distribution kit.

A device driver written in MACRO–11 is divided into several discrete sections:

- Copyright page
- Module header
- Functional description
- Local macro definitions
- Externally defined symbol defaults
- Private and/or local data and symbol definitions, including:
  - Impure area common to all copies of the driver
  - Pure data tables
- An initialization process that creates the required data structures, initializes impure areas, and starts the controller processes
- An impure area that contains state information for the controller processes

- A constant data area that contains constants and tables
- One or more controller processes that wait for I/O requests, activate the device, and wait for a response or a timeout
- One or more interrupt service routines that respond to interrupts from the device
- One or more fork routines that signal the controller processes when the device requires further attention
- A reply routine that returns completion status—successful completion or error code—to the process that requested the I/O operation
- A termination procedure that deallocates owned resources previously obtained by the controller process before deleting the controller process
- Error-processing routines that:
  - Retry on recoverable errors
  - Report nonrecoverable errors to the requesting process
  - Return to the pool I/O request packets that cannot be returned to the requester

Driver components are discussed in detail in the following sections.

14.4.1 Copyright Page

Although optional, the copyright page is recommended if your source module distribution is to be controlled or if it contains proprietary information. Refer to source files contained in your MicroPower/Pascal distribution kit for examples of DIGITAL's copyright statement. Refer to your legal consultant for the wording of your own copyright statement.

At the top of the copyright page—first entry in the source module—include .title and .ident directives. These direct MACRO-11 to place correct headings at the top of each assembly listing page, including source module name and version. For example:

```
.title  xxDRV.MAC - Widget driver
.ident  /V02.00/
```

All information, except the .title and .ident directives, is in the form of comments. That is, each line of text is preceded by a semicolon (;).

14.4.2 Module Header

The module header contains the module name—for example, xxDRV.MAC—system, or application, name, author, creation date, and a brief history of source modifications. Although the information contained in the module header is optional, it is a convenient place to track changes to the source module during the development cycle.

14.4.3 Functional Description

The functional description is also optional. All information in this section is in the form of comment lines. This section contains documentation on how other modules interface with the driver as I/O requests are processed and any other design interface information needed for using the device driver in an application.

14.4.4 Declarations

Declarations include system macro requirements, local macro definitions, external symbol definitions—global symbols defined in the driver prefix module—and local symbol definitions, as required. The following sections describe the declarations.

14.4.4.1 Local Macro Definition

This section contains all macro definitions for local macros referenced by the device driver. (Local macros are referenced only by the device driver proper.)

14.4.4.2 Externally Defined Symbols

The section of externally defined symbols may include symbols containing default values. You need the macros listed when assembling the device driver module. Default values for the macros can be changed, as required, by device driver code. For example, a macro may state the number of retries for read request processing. The particular device driver may also process read-with-no-retries requests, requiring the default value modification.

14.4.4.3 Process Definition

The device driver source must define the controller A process as a static process and, if multiple controllers are supported, request a dynamic process. Defining controller A as a static process is done by issuing the DFSPC$ kernel primitive (described in Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual*). All parameters except ini are required. The following example is the process definition for the DY device driver:

```
DFSPC$ pid=$DYADR,pri=DY$IPR,cxo=0,typ=PT.DRV,grp=1,ter=DYSTP,
cxl=0,sti=$DYAG2,stl=$DYAG1,sth=$DYAG2,start=DYINIT,ini=0
```

Parameter	Definition
pid	$DYADR, the name of the static process (DY driver, controller A)
pri	DY$IPR, the initialization-process priority specified in the driver prefix module
cxo	0—no predefined bit-mask symbols defining hardware context
typ	PT.DRV, indicating device driver (DRIVER) mapping
grp	1, the exception-handling group code of which the process is a member
ter	DYSTP, the DY device driver termination routine entry point
cxl	0 (null parameter)
sti	$DYAG2, the address of the high guard word for the stack; the address loaded into the SP when the process starts
stl	$DYAG1, the address of the low stack guard word; specifies the lower boundary of the stack
sth	$DYAG2, the address of the high stack guard word; specifies the upper boundary of the stack definition macro, DYISZ$
start	DYINIT, the initialization-process entry address
ini	0 (null parameter)

The driver must request a dynamic process if the device driver supports multiple controllers and if the common device driver initialization routine $DDINI is used in the driver's initialization process. You request a dynamic process by issuing the CRPC$P kernel primitive (described in Chapter 3 of the *MicroPower/Pascal Run-Time Services Manual*), immediately preceded by the label $xxPCD::. The following example is the dynamic process request for the DY device driver:

```
$DYPCD::
CRPC$P pdb=PDB,pri=DY$PPR,cxo=0,grp=1,ter=DYSTP,cxl=0,sti=0,
stl=0,sth=0,start=DYSTRT,ini=0
```

Parameter	Definition
pdb	PDB, the argument block address
pri	DY$PPR, the controller process priority specified in the driver prefix module
cxo	0—no predefined bit-mask symbols defining hardware context
grp	1, the exception-handling group code of which the process is a member
ter	DYSTP, the entry point for the termination routine
cxl	0 (null parameter)
sti	0 (no source of initial stack address specified); $DDINI will supply the proper value for this parameter

Parameter	Definition
stl	0 (no source of low boundary address of process stack); $DDINI will supply a value for this parameter
sth	0 (no source of high boundary address of process stack); $DDINI will supply a value for this parameter
start	DYSTRT, the controller process entry address—typically the address immediately following the $DDINI call
ini	0 (null parameter)

A reference to $DYPCD:: is generated by the DRVCF$ macro in the DY driver prefix file; that reference causes the DY driver to be brought in from the driver object library during the application build. See the DRVCF$ macro description in Section 14.2.2.

Sections 14.4.5 and 15.2.3 describe the $DDINI subroutine.

14.4.4.4 Impure-Area Definition

The impure area is the writable data area for the device driver. Each copy of the controller process has its own impure area. Additional space may be allocated in a section according to the number of drivers supported by the controller process. You can see typical contents of the impure area by examining the DY device driver source. Note that the request semaphore SDB is the first item in the impure area and that the ISR impure area is also contained within the impure-area definition.

14.4.4.5 Pure-Area Definition

The pure-area definition is a read-only area that is shared by all driver processes. This area typically contains text, tables, and data that never require modification. In a mapped system, the protection for this area is read-only. For example, in the DY device driver, this area includes a function-code verification table, a device characteristics table, and a table of error-return codes common to all controller processes.

14.4.5 Initialization Process

The initialization process typically creates one or more queue semaphores (by which the requesting processes communicate with the device driver), starts the driver processes for each controller, and initializes the impure area. The driver initialization process is a statically-defined process that executes at very high priority at start-up time. Priority value 250 (decimal) is recommended for start-up initialization. A range of priorities permits specific initialization processes to execute before others, as required.

A common device driver initialization routine, $DDINI, simplifies initialization-process coding for drivers written in MACRO–11. (See Section 15.2.3.) This routine creates the queue semaphore for each controller process and clears the controller process impure area; $DDINI creates the queue semaphore structure descriptor block for controller $xxc in the first 12 bytes of the impure area. If there are multiple controllers in the application, $DDINI creates additional copies of the controller process as needed.

For example, the initialization procedure in the DY device driver performs its functions by calling the $DDINI routine. The required input data is contained in $DYPUR and a pointer to $DYPUR is passed in R5. ($DYPUR is as described in Section 14.2.2.)

```
DYINIT: MOV     #$DYPUR,R5  ; -> Configuration data for RX02
        CALL    $DDINI      ; Common device driver initialization
                            ; routine

        ... [Subtitle and comments omitted -- see Appendix D]

DYSTRT: ...
```

$DDINI exits with the stack containing the controller ID and pointers to the impure area and initialization data, as shown in Section 15.2.3. The controller processes, normally entered immediately following $DDINI execution, must, immediately on entry, remove this controller-specific information from the stack for use by the driver code.

14.4.6 Controller Process

The controller process performs the following functions:

- Reads driver requests
- Validates the requests
- Initiates the I/O functions
- Waits for I/O completion or device timeout and returns status to the requester

There is usually at least one controller process executing for each controller.

You can save some memory by having the initialization process become the controller process rather than create a copy of the driver process. This is done by lowering the process priority to that of the controller process. In the case of multiple driver processes, the initialization process becomes the first driver process and creates copies for each additional process. ($DDINI performs this function.)

The DY device driver controller process includes several routines and subroutines that prepare the driver for I/O, receive and validate I/O requests (queue server), and execute the I/O requests. An overview of the major routines and their functions is provided below.

The DY driver controller process entry point is DYSTRT. Upon entry, the controller process removes and saves the pointers passed by $DDINI; however, it discards the controller ID. The controller process then creates an unnamed binary semaphore that will connect the driver's ISR to the I/O request service process and will connect the interrupt vector to the ISR.

The DYREQ (Request Process Queue Server) routine and three subroutines—DOREQ (Verify and Begin Request), DYTRAN (Start Transfer or Retry), and DYDONE (Finish Processing Request)—contain the device driver's main request-processing code. Those routines perform the following functions:

- Sets up retry count for request
- Validates unit number specified in request; returns an error message if invalid
- Checks function code specified in request; returns an error message if invalid
- Copies device characteristics to the queue element for message returned to the process requesting the I/O operation

- Performs device-specific operations and calls the interrupt procedure (subroutine)
- Returns an appropriate completion status message to the process requesting the I/O operation once the I/O operation is completed

Other subroutines include: INWAIT (Start Function and Wait for Interrupt from Floppy), DOSILO (Initiate Silo Fill or Empty Command), DOXFER (Start Sector Read or Write), DYDOFN (Start Transfer or Silo Operation), DYERR (Error Handler), and REPLY (Return Status Message). (The DY driver's ISR routine and termination procedure are described in Sections 14.4.7 and 14.4.10, respectively.) Those routines are called, as appropriate, when processing read, write, and other operations with the RX02 hardware.

14.4.7 Interrupt Service Routine (ISR)

This section gives a brief overview of the interrupt service routine (ISR). A more detailed description of interrupts, kernel interrupt dispatching, and the ISR is provided in Chapter 7 of the *MicroPower/Pascal Run-Time Services Manual*. Carefully read that chapter before designing or writing ISRs.

Interrupt service routines process the interrupt requests issued by the device hardware. ISRs have the following characteristics:

- Very limited context
- Very brief execution periods
- Restricted processing capabilities

In a mapped system, the mapping of an ISR uses the kernel PARs and is designed to be very fast. Kernel PARs 2 and 3 are saved and then set up to map the driver/ISR code and data. The rest of the mapping context is that of the kernel—the I/O page (PAR 7), system common (PARs 4 through 6, as required), and the kernel code (PARs 0 and 1). See the ISR mapping diagram in Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual*.

To map the ISR code and data via kernel PARs 2 (code) and 3 (data), specify PT.DRV (driver process mapping) for the TYP parameter in the DFSPC$ macro invocation (see Section 14.4.4.3). At application-build time, relocate the driver/ISR code and data virtual addresses to fall within the PAR 2 and PAR 3 address ranges, respectively. You do this by specifying the /O:40000/X (RT host) or /RO:40000/AL (RSX/VMS host) option in the RELOC utility command string. (See the *MicroPower/Pascal–RT System User's Guide* or the *MicroPower/Pascal–RSX/VMS System User's Guide* for a detailed explanation of RELOC utility options.)

Note

The preceding paragraph assumes that the driver process size does not exceed 8K words. As long as the process does not exceed 8K words, the driver and ISR can be mapped according to the driver/ISR mapping conventions outlined in Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual*, and a non-PIC (faster-executing) ISR can be used. DIGITAL recommends that you follow the driver/ISR mapping conventions if at all possible. When the driver size exceeds 8K words, you must use device-access or privileged-process mapping for the driver process, and the ISR must be PIC.

When an ISR is called, the kernel sets the CPU priority to the value specified in the CINT$ primitive issued by the device driver. Since other interrupts at this level and lower cannot occur during interrupt-level processing, the amount of time spent executing ISR code must be minimized.

Before issuing any kernel primitive, the ISR must go to fork-level processing. This is done by issuing a FORK$ call. Once the ISR is at fork level, the CPU priority is set to 0, permitting other interrupts to be serviced.

A very brief ISR is contained in the DY device driver ($DYINT). Upon entry, the ISR immediately issues a FORK$ request. Once at fork level, the ISR restores registers, checks for errors, and JMPs the routine that initiated the RX02 function. Since the RX02 is a DMA device, all data transfers have been completed, and the interrupt simply informs the device driver of that fact. The DY driver is listed in Appendix D.

Other device drivers require more extensive ISRs. A more extensive ISR might perform I/O or queue operations internally.

14.4.8 Fork Routine

An ISR issues a FORK$ call to exit processing at interrupt level and to enter fork-level processing whenever the ISR must issue a kernel primitive request—for example, in order to signal the controller process. Kernel primitives cannot be issued at interrupt level.

The code following the FORK$ call is the fork routine and is generally shown as part of the ISR. When at fork level, the fork routine continues execution with ISR mapping, but at a CPU priority of 0; thus, other interrupts, including lower-priority interrupts, can be serviced. Only ISR routines, interrupted kernel primitive execution, and other fork routines are executed at fork level; all fork routines are completed before returning to the interrupted process level.

14.4.9 Reply Subroutine

The reply subroutine returns completion status to the process that requested the I/O operation. Status indicates either successful completion or an appropriate error code. DIGITAL supplies the $DRPLY (Send Device Driver Reply) subroutine, described in Section 15.2.8, for that purpose.

The caller of the reply subroutine inserts the status code, error code, and byte count (actual length) involved in the I/O operation in the reply queue element (message packet). If this is deferred until just before the reply, the driver can modify the original request packet and return it as a reply packet—rather than allocating a separate reply packet.

Upon entry, the reply subroutine should receive a pointer to the message packet.

The DY driver REPLY (Return Status Message) routine sets up and executes a call to the $DRPLY subroutine.

14.4.10 Termination Procedure

Each device driver includes a termination procedure that is entered whenever the device driver is stopped (aborted). The termination procedure deallocates, in an orderly manner, all resources that the driver had previously acquired. The termination procedure performs the following functions:

- Disables interrupts on the appropriate controller hardware
- Cancels any active timeout requests
- Returns all outstanding requests to the requesting process with abort status, if possible; otherwise, the termination procedure returns the request packet(s) to the kernel pool
- Destroys all structures created by the driver
- Deletes all processes created by the driver
- Deletes the driver controller process

For an example of a termination procedure, refer to the DY driver DYSTP (Request Process Termination) routine. DYSTP disables interrupts, returns abort status to waiting processes, deletes all structures, and finally deletes the driver process.

14.4.11 Error-Processing Routines

The driver processes several categories of errors, as follows:

- Invalid request packets
- Exceptions (traps—memory timeouts, illegal instructions, and so forth)
- Drive or controller hardware errors
- Resource famine (insufficient kernel pool to allocate structures, and so forth)

14.4.11.1 Invalid Requests

Drivers are responsible for validating I/O request packets as completely as possible. Processing an invalid request as though it were valid can corrupt a system; every attempt must be made to prevent that from happening. If possible, the driver returns the invalid packet to the sender.

14.4.11.2 Exceptions

Exceptions are memory timeouts, illegal instructions, traps, and so forth, as follows:

- Invalid parameters in the I/O request packet—for example, a request to write into read-only memory or a request to access nonexistent memory
- One or more programming errors in the device driver
- A hardware failure—CPU error, memory error, or controller error
- Nonexistent hardware

Device drivers must validate all the I/O request packets to prevent exceptions caused by invalid parameters. Thorough testing will minimize exceptions caused by programming errors. Hardware errors should be anticipated so their occurrence does not cause unexpected run-time errors that corrupt memory contents or produce other unpredictable results. The device-initialization process must detect exceptions caused by nonexistent hardware in order to avoid unexpected run-time errors.

Applications should include exception handlers for all types of exceptions that may be caused by the processing of bad data in a request packet. For example, if an incorrect address supplied in the request packet causes the driver to generate a memory fault (trap to 4), the driver should provide a memory fault exception handler for the problem. (Exception handling and dispatching are discussed in Chapter 6 of the *MicroPower/Pascal Run-Time Services Manual*.)

14.4.11.3 Drive or Controller Errors

Drive or controller errors can be either recoverable or nonrecoverable. In the case of recoverable errors, the device driver should retry eight times before considering the error nonrecoverable, unless the requesting process has specified in the I/O request packet that retries are disabled. In the case of nonrecoverable errors, the device driver returns an appropriate error status to the requesting process.

14.4.11.4 Resource Famine

If a standard device driver or the common initialization routine $DDINI cannot obtain enough memory to create a process or structure, it reports an exception by calling the common exception-reporting routine $DDEXC (see Section 15.2.2), which issues an $REXC kernel primitive request. Resource famine errors should occur only during the debugging stages of application program development.

Chapter 15
Device Driver Macros and Subroutines

This chapter describes macros and subroutines that can be used by device drivers written in MACRO-11.

15.1 Driver Macros

The following driver macros are available from the kernel macro libraries COMM and COMU for the use of user-written device drivers:

Macro	Description
ADPAR$	Accept virtual address and return the address of corresponding kernel PAR
DRMAP$	Remap Virtual Address
DRPAR$	Read Contents of PAR or PDR Register
DRVDF$	Define Driver Packet Symbols
DSCXW$	Disable MMU Context Switch
DWPAR$	Write to PAR or PDR Register
ENCXW$	Enable MMU Context Switch
IBADR$	Increment Byte Address and Check for PAR Tick Overflow
IWADR$	Increment Word Address and Check for PAR Tick Overflow
MVBYT$	Move Byte from/to Virtual Addresses
MVBYU$	Move Byte from/to Virtual Addresses from user-mode
MVMAP$	Move Word from/to Virtual Addresses in mapped case only
MVVAD$	Move Address and PAR

Macro	Description
MVWRD$	Move Word from/to Virtual Addresses
MVWRU$	Move Word from/to Virtual Addresses from user-mode
SPL$	Set Priority Level
XTAD$	Compute Bus Extended Address

To use a driver macro, you list the macro name in an .MCALL statement in your driver source file and then invoke the macro in accordance with the syntax shown in this chapter. In addition, if you assemble the driver by manually invoking the assembler (rather than letting MPBUILD do it for you), you must include in the assembler command line the COMM or COMU macro library and the appropriate qualifier (for example, "/ML").

The purpose of many of the driver macros is to allow a driver to be written for both mapped and unmapped systems without conditional assembly. Where the mapped (COMM) versions of the macros generate code that manipulates virtual addresses and PAR values, the unmapped (COMU) versions generate shorter code sequences, with no PAR references, or no code at all.

Note

Driver prefix file macros and system configuration macros are not included in this section. The driver prefix file macros, such as DRVCF$ and CTRCF$, are described in Chapter 14. The system configuration macros, including the driver-related DEVICES macro, are described in the *MicroPower/Pascal Run-Time Services Manual*.

15.1.1 ADPAR$ (Return PAR Address)

The ADPAR$ macro accepts a virtual address and returns the address of the corresponding kernel PAR. This macro may be useful if you want the driver to treat part of the request packet as a standard user buffer (with a virtual address and a PAR value) in common driver code. You can use either the kernel PAR or the driver PAR because the request packet is in the kernel impure area.

Syntax

```
ADPAR$ viradr,paradr
```

Parameter	Definition
viradr	The virtual address you enter. This argument has the form: `[VIRADR=]virtual-address`
paradr	The location to receive the PAR address. This argument has the form: `[PARADR=]PAR address`

Semantics

In the mapped case, the ADPAR$ macro generates:

```
MOV     R0,-(SP)
MOV     viradr,R0
ASH     #-13.,R0
BIC     #^C7,R0
ASL     R0
ADD     #K.ISA0,R0
MOV     R0,paradr
MOV     (SP)+,R0
```

No code is generated in the unmapped case.

15.1.2 DRMAP$ (Remap Virtual Address)

The DRMAP$ macro converts a virtual address and its associated PAR value into an address and PAR value that are usable in the caller-specified PAR. The resulting address is adjusted to the lower boundary of the specified PAR. For example, DRMAP$ allows drivers and ISRs to convert the buffer address and PAR value received for an I/O request in order to use PAR 1 (which the kernel-mode ISR must save and restore) during data transfers. The DRMAP$ macro generates a call to the $DRMAP routine that resides in the mapped driver object library DRVM.

All registers are preserved across the call.

Syntax

```
DRMAP$ srcadr,srcpar,dstadr,dstpar,parnum
```

Parameter	Definition
srcadr	The address to remap. This argument has the form: `[SRCADR=]source-address`
srcpar	The PAR value for SRCADR. This argument has the form: `[SRCPAR=]source-par`
dstadr	The location to receive the remapped address. This argument has the form: `[DSTADR=]destination-address`
dstpar	The location to receive the remapped PAR value. This argument has the form: `[DSTPAR=]destination-par`
parnum	The PAR number to map to. The default is #1. This argument the form: `[PARNUM=]par-number`

Semantics

In the mapped case, the DRMAP$ macro generates the following code:

```
MOV     srcpar,-(SP)
MOV     srcadr,-(SP)
MOV     parnum,-(SP)
.GLOBL  $DRMAP
JSR     PC,$DRMAP
MOV     (SP)+,dstadr
MOV     (SP)+,dstpar
```

In the unmapped case, the macro generates:

```
MOV     srcadr,dstadr
```

Example

The following NSP source excerpt transfers user-requested read data to a user-specified buffer and then replies to the user.

R0 contains the count of bytes to be transferred, R1 points to the start of the data to be copied, and R4 points to the user request packet (also used for the reply). The buffer to receive the read data is specified by the DP.BUF and DP.PAR fields of the request packet.

Other symbols for the example are defined by macros DRVDF$ (DP.xxx packet offsets), PCBDF$ (PCB symbols for ENCXW$ and DSCXW$), and IODF$ (U.ISA1 and U.ISD1, the user PAR 1 and PDR 1 I/O page addresses). There are global symbols, U.ISA0–U.ISA7, that define the I/O page addresses of user PARs 0–7, respectively. There are also global symbols U.ISD0–U.ISD7 that define the I/O page addresses of user PDRs 0–7, respectively. Similarly, K.ISA0–K.ISA7 and K.ISD0–K.ISD7 define the I/O page addresses of kernel PARs 1–7 and kernel PDRs 1–7, respectively.

In the excerpt, DRMAP$ maps the user-specified virtual-address/PAR pair to PAR 1 for the transfer of the read data. The ENCXW$, DWPAR$, and DSCXW$ macros and the $BLXIO and $DRPLY subroutines are described in detail elsewhere in this chapter.

```
60$:    MOV     R0,DP.ALN(R4)   ;This is the actual length transferred
        BEQ     70$             ;Could be zero length message
        ENCXW$  pcb=@#$RUN      ;Turn on MMU context switching now
        DWPAR$  #77406,U.ISD1   ;Initialize PDR 1 to full access
        DRMAP$  srcadr=DP.BUF(R4),srcpar=DP.PAR(R4),
        dstadr=R2,dstpar=@#U.ISA1,parnum=#1
        CALL    $BLXIO          ;Copy R0 bytes from (R1) to (R2)
        DSCXW$  pcb=@#$RUN      ;Turn off MMU context switching now
70$:    CALL    $DRPLY          ;Give response to user
```

15.1.3 DRPAR$ (Read Contents of PAR or PDR Register)

The DRPAR$ macro fetches the contents of a PAR or PDR register.

Syntax

```
DRPAR$ parnam,where
```

Parameter	Definition
parnam	The PAR to be read. This argument has the form: `[PARNAM=]par-name`
where	The destination for the PAR contents. This argument has the form: `[WHERE=]destination-address`

Semantics

In the mapped case, the DRPAR$ macro generates:

```
MOV   @#parnam,where
```

No code is generated in the unmapped case.

Example

```
DRPAR$   U.ISA1,-(SP)    ;Save the contents of user PAR 1
                         ;on the stack
```

15.1.4 DRVDF$ (Define Driver Packet Symbols)

The DRVDF$ macro defines symbols required for packet-level use of the MicroPower/Pascal device drivers, including:

- I/O packet field offsets (DP.xxx)
- Function codes (IF$xxx)
- Function modifiers (FM$xxx)
- Device class codes (DC$xxx) and type codes

The DRVDF$ symbols are listed in Chapter 1 and referenced throughout this manual.

Syntax

```
DRVDF$ range
```

Parameter	Definition
range	A parameter that determines if the symbols are global. The default is nogbl for not global. This argument has the form: [RANGE=]nogbl or [RANGE=]global

15.1.5 DSCXW$ (Disable MMU Context Switch)

The DSCXW$ macro disables MMU context switching (CX$KT) for a process control block (PCB). It is used with the ENCXW$ (Enable MMU Context Switch) macro, described in Section 15.1.7.

The ENCXW$ and DSCXW$ macros are used by drivers that are actively accessing a user buffer (implying non-DMA transfers) from user mode. The two macros allow you to reduce MMU-register-saving overhead by enabling MMU context switching only when necessary.

Setting CX$KT causes the kernel to save/restore the MMU registers—a lengthy sequence—when the driver blocks/resumes. Since the drivers block frequently—waiting for the next user request—repeated saving/restoring of the MMU registers occurs. To reduce that overhead, use ENCXW$ and DSCXW$ to bracket code that modifies mapping to access the user buffer, leaving MMU context switching disabled when not needed. (See the example in Section 15.1.2.)

Use of the ENCXW$ and DSCXW$ macros requires a firm grasp of MMU context switching.

DSCXW$ requires DRIVER or PRIVILEGED mapping, because the generated code accesses the PCB. You must also define PCB symbols by invoking the PCBDF$ macro.

All registers are preserved across the macro call, except for a user-specified work register (if provided).

Syntax

```
DSCXW$ pcb[,reguse]
```

Parameter	Definition
pcb	The address of the PCB for which MMU context switching is to be disabled. This argument has the form: `[PCB=]pcb-address`
reguse	An optional parameter that specifies a register to use for the operation; if not specified, extra instructions that save and restore a register are generated. This argument has the form: `[REGUSE=]register-spec`

Semantics

In the mapped case, if the REGUSE parameter is not provided, the ENCXW$ macro generates the following code:

```
MOV   R0,-(SP)
MOV   pcb,R0
BICB   #CX$KT,PC.CXW(R0)
MOV   (SP)+,R0
```

If mapped, and REGUSE is provided, the following is generated:

```
MOV   pcb,reguse
BICB   #CX$KT,PC.CXW(reguse)
```

No code is generated in the unmapped case.

Application Note

A space improvement can be made to any driver that keeps a copy of its process descriptor (filled in by means of a GTST$ kernel primitive call) only for the purpose of supplying the driver's PCB address to ENCXW$ and DSCXW$. The kernel symbol $RUN can be used instead to supply the current PCB address (DSCXW$ pcb=@#$RUN...), thereby potentially eliminating the driver's need for the copy of the PDB and the services of the GTST$ kernel primitive.

15.1.6 DWPAR$ (Write to PAR or PDR Register)

The DWPAR$ macro loads a PAR or PDR register.

The example in Section 15.1.2 illustrates the use of DWPAR$.

Syntax

```
DWPAR$ from,parnam
```

Parameter	Definition
from	The address from which data will be loaded. This argument has the form: `[FROM=]source-address`
parnam	The PAR to be loaded. This argument has the form: `[PARNAM=]par-name`

Semantics

In the mapped case, the DWPAR$ macro generates:

```
MOV   from,@#parnam
```

No code is generated in the unmapped case.

Example

```
DWPAR$   #77406,U.ISD1   ;Set PDR 1 for full RW access
```

15.1.7 ENCXW$ (Enable MMU Context Switch)

The ENCXW$ macro enables MMU context switching (CX$KT) for a process control block (PCB). It is used with the DSCXW$ (Disable MMU Context Switch) macro, described in Section 15.1.5.

The ENCXW$ and DSCXW$ macros are used by drivers that are actively accessing a user buffer (implying non-DMA transfers) from user mode. The two macros allow you to reduce MMU-register-saving overhead by enabling MMU context switching only when necessary.

Setting CX$KT causes the kernel to save/restore the MMU registers—a lengthy sequence—when the driver blocks/resumes. Since the drivers block frequently—waiting for the next user request—repeated saving/restoring of the MMU registers occurs. To reduce that overhead, use ENCXW$ and DSCXW$ to bracket code that modifies mapping to access the user buffer, leaving MMU context switching disabled when not needed. (See the example in Section 15.1.2.)

Use of the ENCXW$ and DSCXW$ macros requires a firm grasp of MMU context switching.

ENCXW$ requires DRIVER or PRIVILEGED mapping, because the generated code accesses the PCB. You must also define PCB symbols by invoking the PCBDF$ macro.

All registers are preserved across the macro call, except for a user-specified work register (if provided).

Syntax

```
ENCXW$ pcb[,reguse]
```

Parameter	Definition
pcb	The address of the PCB for which MMU context switching is to be enabled. This argument has the form: `[PCB=]pcb-address`
reguse	An optional parameter that specifies a register to use for the operation; if not specified, extra instructions that save and restore a register are generated. This argument has the form: `[REGUSE=]register-spec`

Semantics

In the mapped case, if the REGUSE parameter is not provided, the ENCXW$ macro generates the following code:

```
MOV   R0,-(SP)
MOV   pcb,R0
BISB   #CX$KT,PC.CXW(R0)
MOV   (SP)+,R0
```

If mapped, and REGUSE is provided, the following is generated:

```
MOV   pcb,reguse
BISB   #CX$KT,PC.CXW(reguse)
```

No code is generated in the unmapped case.

Application Note

A space improvement can be made to any driver that keeps a copy of its process descriptor (filled in by means of a GTST$ kernel primitive call) ONLY for the purpose of supplying the driver's PCB address to ENCXW$ and DSCXW$. The kernel symbol $RUN can be used instead to supply the current PCB address (ENCXW$ pcb=@#$RUN...), thereby potentially eliminating the driver's need for the copy of the PDB and the services of the GTST$ kernel primitive.

15.1.8 IBADR$ (Increment Byte Address and Check for PAR Tick Overflow)

The IBADR$ macro increments a virtual byte address that has been remapped via the $DRMAP macro to point to the next byte. It also checks for overflow of the PAR field and adjusts the address and the PAR accordingly.

Since the result of $DRMAP is an address adjusted to the lower boundary of a PAR, simply incrementing the address is safe until approximately 8K–64 bytes have been transferred. If there is no need to handle such large transfers, you do not need to use this macro.

Syntax

```
IBADR$ addr,par,?skip
```

Parameter	Definition
addr	The byte address that was previously remapped via DRMAP$. This argument has the form: `[ADDR=]byte-address`
par	The PAR value associated with the byte address. This argument has the form: `[PAR=]par-value`
?skip	A label name for use by a branch instruction in the generated code. This argument has the form: `[?SKIP=]label-name`

Semantics

In the mapped case, the IBADR$ macro generates the following code:

```
INC   addr
BITB  #77,addr
BNE   skip
SUB   #100,addr
INC   par
skip:
```

In the unmapped case, the macro generates:

```
INC   addr
```

15.1.9 IWADR$ (Increment Word Address and Check for PAR Tick Overflow)

The IWADR$ macro increments a virtual word address that has been remapped via the $DRMAP macro to point to the next word. It also checks for overflow of the PAR field and adjusts the address and the PAR accordingly.

Since the result of $DRMAP is an address adjusted to the lower boundary of a PAR, simply incrementing the address is safe until approximately 8K–64 bytes have been transferred. If there is no need to handle such large transfers, you do not need to use this macro.

Syntax

```
IWADR$ addr,par,?skip
```

Parameter	Definition
addr	The word address that was previously remapped via DRMAP$. This argument has the form: `[ADDR=]word-address`
par	The PAR value associated with the word address. This argument has the form: `[PAR=]par-value`
?skip	A label name for use by a branch instruction in the generated code. This argument has the form: `[?SKIP=]label-name`

Semantics

In the mapped case, the IBADR$ macro generates the following code:

```
ADD     #2,addr
BITB    #77,addr
BNE     skip
SUB     #100,addr
INC     par
skip:
```

In the unmapped case, the macro generates:

```
ADD     #2,addr
```

15.1.10 MVBYT$ (Move Byte from/to Virtual Addresses)

The MVBYT$ macro allows an ISR to move a byte of data from and to caller-specified virtual addresses. A caller-specified PAR value is used to map the user data area with kernel APR 1. Kernel PAR 1 is saved before and restored after this operation.

MVBYT$ is used with the DRMAP$ (Remap Virtual Address) and IBADR$ (Increment Byte Address and Check for PAR Tick Overflow) macros. Specifically, you use DRMAP$ to adjust a virtual-address/PAR pair to a PAR 1 virtual-address/PAR pair (specifying PARNUM=#1) in order to set up the MVBYT$ (or other data-moving operation).

MVBYT$ works only in kernel mode. Fork routines should not use MVBYT$.

Syntax

```
MVBYT$ src,dst,par
```

Parameter	Definition
src	The source address. This argument has the form: `[SRC=]source-address`
dst	The destination address. This argument has the form: `[DST=]destination-address`
par	The PAR value used to map the user data area. This argument `[PAR=]par-value`

Semantics

In the mapped case, the MVBYT$ macro generates the following code:

```
MOV   @#K.ISA1,-(SP)
MOV   par,@#K.ISA1
MOVB   src,dst
MOV   (SP)+,@#K.ISA1
```

In the unmapped case, the macro generates:

```
MOVB   src,dst
```

15.1.11 MVBYU$ (Move Byte from/to Virtual Addresses from User-Mode)

The MVBYU$ macro moves a byte from the source to the destination in the unmapped case. In the mapped case, it optionally saves the user PAR 1 value, maps user APR 1 using the value par, and moves a byte. It then optionally restores user PAR 1. This macro only works in user mode; that is, from driver processes. ISRs should not use MVBYU$.

The calling process should enable MMU context switching before invoking this macro and disable it after invoking the macro. The DCSXW$ and ENCXW$ macros are provided for this purpose.

Syntax

```
MVBYU$   src,dst,par,savmap,resmap
```

Parameter	Definition
src	The source address. This argument has the form: `[SRC=]source-address`
dst	The destination address. This argument has the form: `[DST=]destination-address`
par	The PAR value used to map the user data area. This argument has the form: `[PAR]par-value`
savmap	The saving of the mapping registers. The default is NO. This argument has the form: `SAVMAP=YES or NO`
resmap	The restoring of the mapping registers. The default is NO. This argument has the form: `RESMAP=YES or NO`

Semantics

In the mapped case, with SAVMAP and RESMAP both defaulted to NO, the MVBYU$ macro generates:

```
MOV   par,@#U.ISA1
MOV   #77406,@#U.ISD1
MOVB   src,dst
```

In the unmapped case, the MVBYU$ macro generates:

```
MOVB   src,dst
```

15.1.12 MVMAP$ (Move Word from/to Virtual Addresses in Mapped Case Only)

The MVMAP$ macro moves a word from the source to the destination in the mapped case only.

Syntax

```
MVMAP$   src,dst
```

Parameter	Definition
src	The source address. This argument has the form: `[SRC=]source-address`
dst	The destination address. This argument has the form: `[DST=]destination-address`

Semantics

In the mapped case, the MVMAP$ macro generates:

```
MOV   src,dst
```

No code is generated in the unmapped case.

15.1.13 MVVAD$ (Move Address and PAR)

The MVVAD$ macro moves an address and the associated PAR to another pair of words.

Syntax

```
MVVAD$ srcadr,srcpar,dstadr,dstpar
```

Parameter	Definition
srcadr	The address to be moved. This argument has the form: `[SRCADR=]address`
srcpar	The PAR value to be moved. This argument has the form: `[SRCPAR=]par-value`
dstadr	The address destination. This argument has the form: `[DSTADR=]destination-address`
dstpar	The PAR destination. This argument has the form: `[DSTPAR=]destination-par`

Semantics

In the mapped case, the MVVAD$ macro generates the following code:

```
MOV   srcadr,dstadr
MOV   srcpar,dstpar
```

In the unmapped case, the macro generates:

```
MOV   srcadr,dstadr
```

15.1.14 MVWRD$ (Move Word from/to Virtual Addresses)

The MVWRD$ macro allows an ISR to move a word of data from and to caller-specified virtual addresses.

A caller-specified PAR is used to map the user data area with kernel APR 1.

MVWRD$ is used with the DRMAP$ (Remap Virtual Address) and IWADR$ (Increment Word Address and Check for PAR Tick Overflow) macros. Specifically, you use DRMAP$ to adjust a virtual-address/PAR pair to a PAR 1 virtual-address/PAR pair (specifying PARNUM=#1) in order to set up the MVWRD$ (or other data-moving operation).

MVWRD$ works only in kernel mode. Fork routines should not use MVWRD$.

Syntax

```
MVWRD$ src,dst,par
```

Parameter	Definition
src	The source address. This argument has the form: `[SRC=]source-address`
dst	The destination address. This argument has the form: `[DST=]destination-address`
par	The PAR value used to map the user data area. This argument has the form: `[PAR=]par-value`

Semantics

In the mapped case, the MVWRD$ macro generates the following code:

```
MOV   @#K.ISA1,-(SP)
MOV   par,@#K.ISA1
MOV   src,dst
MOV   (SP)+,@#K.ISA1
```

In the unmapped case, the macro generates:

```
MOV   src,dst
```

15.1.15 MVWRU$ (Move Word from/to Virtual Addresses from User-Mode)

The MVWRU$ macro moves a word from the source to the destination in the unmapped case. In the mapped case, MVWRU$ optionally saves the user PAR 1 value, maps user APR 1 using the value par, and moves the word. It then optionally restores user PAR 1. This macro only works in user mode; that is, from driver processes. ISRs should not use MVWRU$.

The calling process should enable MMU context switching before invoking this macro and disable it after invoking the macro. The DCSXW$ and ENCXW$ macros are provided for this purpose.

Syntax

```
MVWRU$ src,dst,par,savmap,resmap
```

Parameter	Definition
src	The source address. This argument has the form: `[SRC=]source-address`
dst	The destination address. This argument has the form: `[DST=]destination-address`
par	The PAR used to map the user data area. This argument has the form: `[PAR]par-name`
savmap	The saving of the mapping registers. The default is NO. This argument has the form: `SAVMAP=YES or NO`
resmap	The restoring of the mapping registers. The default is NO. This argument has the form: `RESMAP=YES or NO`

Semantics

In the mapped case, if SAVMAP and RESMAP are defaulted to NO, the MVWRU$ macro generates:

```
MOV   par,@#U.ISA1
MOV   #77406,@#U.ISD1
MOV   src,dst
```

In the unmapped case, the MVWRU$ macro generates:

```
MOV   src,dst
```

15.1.16 SPL$ (Set Priority Level)

The SPL$ macro is used by a driver process executing in user mode to raise its hardware priority to 7 and lower it back to 0. It is useful for avoiding race conditions between drivers and their ISRs.

In the mapped case, SPL$ issues a call to the Set Hardware Priority Level ($SPL) kernel primitive, which resides in the mapped kernel object library PAXM. In the unmapped case, it performs a byte operation on the low byte of the PSW.

Note

R0 is destroyed in the mapped case of SPL$ but preserved in the unmapped case.

SPL$ has a potentially dangerous global effect and should be used with caution. In particular, once a driver is at priority 7, it must not issue any primitives (other than $SPL).

SPL$ cannot be used from ISR or fork level. However, ISR or fork routines can use the SPL macro, which resides in the COMM and COMU kernel macro libraries. SPL performs a byte operation on the low byte of the PSW in both the mapped and unmapped cases.

Syntax

```
SPL$ pri[,savreg]
```

Parameter	Definition
pri	The hardware priority level to set; must be 0 or 7. This argument has the form: [PRI=]7 or [PRI=]0
savreg	The specification to save or not save R0; can be YES or NO. The default is NO. This argument has the form: [SAVREG=]YES or [SAVREG=]NO

Semantics

In the mapped case, the SPL$ macro generates the following code:

```
MOV    #<pri*40>,-(SP)
MOV    SP,R0
IOT
.word  $SPL
INC    (SP)+
```

In the unmapped case, the macro generates:

```
MOV    #<pri*40>,-(SP)
MTPS   (SP)+
```

15.1.17 XTAD$ (Compute Bus Extended Address)

The XTAD$ macro computes the extended address bits and the physical bus address from a specified virtual address and PAR value. R0 is destroyed. The XTAD$ macro generates a call to the $XTADR subroutine that resides in the mapped driver object library DRVM.

XTAD$ is primarily used to compute full 22-bit addresses for DMA device drivers. The equivalent Pascal method is shown below.

Syntax

```
XTAD$ vadd,par,pos,ext,addr
```

Parameter	Definition
vadd	The virtual address. This argument has the form: `[VADD=]virt-address`
par	The PAR value. This argument has the form: `[PAR=]par-value`
pos	The bit position, in the EXT argument, at which the low-order bit of the extended address is to be placed. This argument has the form: `[POS=]bit-number`
ext	The target address for the shifted extended address bits. This argument has the form: `[EXT=]address`
addr	The target address for the low-order address bits. This argument has the form: `[ADDR=]address`

Semantics

In the mapped case, if the POS argument is 0, the XTAD$ macro generates the following code:

```
CMP   -(SP),-(SP)
MOV   SP,R0
MOV   par,-(SP)
MOV   vadd,-(SP)
MOV   R0,-(SP)
JSR   PC,$XTADR
MOV   (SP)+,addr
MOV   (SP)+,R0
MOV   R0,ext
```

If mapped and POS is nonzero, the macro generates:

```
CMP   -(SP),-(SP)
MOV   SP,R0
MOV   par,-(SP)
MOV   vadd,-(SP)
MOV   R0,-(SP)
JSR   PC,$XTADR
MOV   (SP)+,addr
MOV   (SP)+,R0
ASH   #pos,R0
MOV   R0,ext
```

In the unmapped case, if the ADDR parameter is not provided, the macro generates:

```
CLR   ext
MOV   vadd,R0
```

If unmapped and ADDR is provided, the macro generates:

```
CLR   ext
MOV   vadd,R0
MOV   R0,ADDR
```

Example of Pascal Equivalent

Given the virtual page base (PARV) and the virtual address (ADDR) of a data item, the following Pascal sequence translates those values into the 22-bit physical address (BUS_ADDRESS) of the data item:

```
VAR
    addr : UNSIGNED;
    parv : UNSIGNED;
    bus_address : LONG_INTEGER;      {Enough for all 22 bits}
    offset : UNSIGNED;
BEGIN
    offset := UAND(addr,%O'17777');   {How far from page base}
    bus_address := parv;      {Conv page base to 32 bits}
    bus_address := (bus_address * 64) + offset; {Construct full addr}
   ...
END;
```

15.2 Driver Subroutines

The following driver subroutines are available from the driver object libraries DRVM and DRVU for the use of user-written device drivers:

Routine	Description
$BLXIO	Block Move
$DDEXC	Report Exception for Device Driver
$DDINI	Device Driver Initialization
$DRALR	Allocate Memory
$DRDSP	Deallocate Dynamic Memory
$DRHIN	Initialize Heap
$DRNEW	Allocate Dynamic Memory
$DRPLY	Send Device Driver Reply
$SV02	Save/Restore Registers 0–2
$SV03	Save/Restore Registers 0–3
$SV05	Save/Restore Registers 0–5

To use a driver subroutine, you reference the subroutine name in a call statement in your driver source file, following any appropriate conventions noted in this chapter for that subroutine. In addition, if you build the driver into your application manually (rather than letting MPBUILD do it for you), you must merge the driver with the DRVM or DRVU driver library by including DRVM or DRVU in the MERGE command line, with the appropriate qualifier (for example, "/LB").

Note

The $DRMAP and $XTADR subroutines are described in Section 15.1 (the driver macros section) rather than this section, because they are accessed with macro calls. See the DRMAP$ and XTAD$ macro descriptions.

15.2.1 $BLXIO (Block Move)

The $BLXIO subroutine moves data from one buffer to another. It assumes that all data is currently mapped within the calling process's virtual address space. Given a source address, destination address, and byte count—that may be even or odd, in any combination—$BLXIO selects the most efficient means of transfer.

$BLXIO works in both mapped and unmapped systems.

The example in Section 15.1.2 illustrates the use of $BLXIO.

Calling Syntax

```
JSR   PC,$BLXIO
```

Input

R0 must contain the number of bytes to be moved, R1 the source address, and R2 the destination address.

Output

Upon return, the data has been copied. R0 has been destroyed, R1 points to the byte after the last byte of the source, and R2 points to the byte after the last byte of the destination.

15.2.2 $DDEXC (Report Exception for Device Driver)

The $DDEXC subroutine reports an exception for a driver. The routine is usually called immediately after a kernel primitive request has returned an error. $DDEXC does not return—except in the unusual case where the driver has an exception handler. After the exception is reported, the calling process is stopped.

$DDEXC works in both mapped and unmapped systems.

Calling Syntax

```
JSR   PC,$DDEXC
```

Input

R0 must contain the exception code to be reported—normally an error code that was returned by a kernel primitive.

15.2.3 $DDINI (Device Driver Initialization)

The $DDINI subroutine performs standard device driver initialization. It creates the queue semaphore for each controller process ($xxc). If there are multiple controllers in the application, $DDINI creates additional copies of the controller process as needed. The impure area for each controller process is cleared.

Registers 0 through 4 are modified.

Any errors detected by $DDINI are reported via the $DDEXC subroutine and are not recoverable.

$DDINI works in both mapped and unmapped systems.

See also the prefix file macros DRVCF$ (Configure Device Driver) and CTRCF$ (Configure Controller), described in Chapter 14.

Calling Syntax

```
JSR   PC,$DDINI
<start-address>:
```

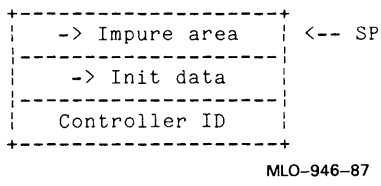
The address of the instruction after the $DDINI call is normally the start address for the calling (first controller) process and for each additional controller process as well. Upon return from $DDINI, the first controller process falls into the start code. Subsequent controller processes are entered at the start address specified by a CRPC$P macro start-address parameter, which normally matches the label on the instruction after the $DDINI call.

Input

R5 contains the address of the driver configuration data area ($xxPUR::). That area is defined by the prefix file macro DRVCF$, described in Chapter 14.

Output

For each controller, the controller ID and pointers to the controller's impure area and initialization data are returned on the stack, as shown below. The controller processes are normally entered immediately following $DDINI execution. Each controller process must remove this controller-specific information from the stack immediately on entry for use by the driver code.



MLO-946-87

The initialization data area (at $xxcPU::) is defined by the prefix file macro CTRCF$, described in Chapter 14.

The controller ID is 0 for controller A, 1 for controller B, and so forth.

15.2.4 $DRALR (Allocate Memory)

The $DRALR subroutine allocates memory from the kernel in a consistent manner, using the $ALRG (Allocate Region) kernel primitive. All registers are preserved across the call.

$DRALR works in both mapped and unmapped systems.

Calling Syntax

```
JSR   PC,$DRALR
```

Input

The size, in bytes, to be allocated is passed on the stack.

Output

If no memory is available, the carry bit is set. Otherwise, offset and PAR values describing the allocated memory are returned on the stack.

Example

```
MOV   #size,-(SP)    ; Supply size in bytes
JSR   PC,$DRALR      ; Call allocate routine
BCS   error          ; Branch if no memory available
MOV   (SP)+,addr     ; Get virtual address
MOV   (SP)+,par      ; and PAR value
```

15.2.5 $DRDSP (Deallocate Dynamic Memory)

The $DRDSP subroutine deallocates heap memory that was allocated with the $DRNEW (Allocate Dynamic Memory) subroutine. ($DRNEW and $DRDSP are analogous to NEW and DISPOSE in Pascal.)

All registers are preserved across a $DRDSP call.

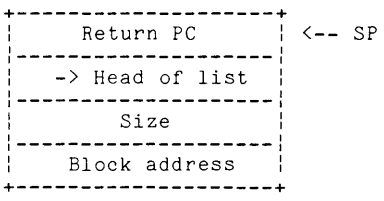
$DRDSP works in both mapped and unmapped systems.

Calling Syntax

```
JSR   PC,$DRDSP
```

Input

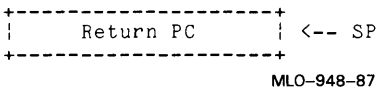
The address of the block of memory to deallocate, the block size in bytes, and a pointer to the head of the free memory list are passed on the stack, as follows:



MLO-947-87

Output

On exit, the stack is as follows:



MLO-948-87

Error Returns

The following error code can be returned in R0, with the carry bit set:

Code	Type	Description
ES$DDP	EX$RSC	DISPOSE of already disposed pointer

15.2.6 $DRHIN (Initialize Heap)

The $DRHIN subroutine initializes a memory heap, which can then be managed with the $DRNEW (Allocate Dynamic Memory) and $DRDSP (Deallocate Dynamic Memory) subroutines.

All registers are preserved across a $DRHIN call.

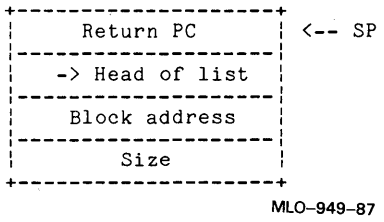
DRHIN$ works in both mapped and unmapped systems.

Calling Syntax

```
JSR   PC,$DRHIN
```

Input

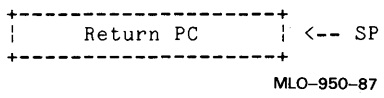
The heap size in bytes, the address of the heap base, and an address to receive a pointer to the head of the free memory list are passed on the stack, as follows:



MLO-949-87

Output

On exit, the stack is as follows:



MLO-950-87

15.2.7 $DRNEW (Allocate Dynamic Memory)

The $DRNEW subroutine allocates a block of memory from a memory heap that has been initialized with the $DRHIN (Initialize Heap) subroutine. ($DRNEW is analogous to NEW in Pascal.)

All registers are preserved across a $DRNEW call.

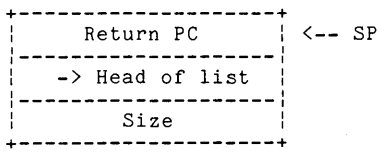
$DRNEW works in both mapped and unmapped systems.

Calling Syntax

```
JSR   PC,$DRNEW
```

Input

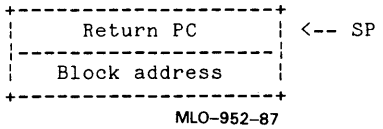
The requested allocation in bytes and a pointer to the head of the free memory list are passed on the stack, as follows:



MLO-951-87

Output

The address of the allocated block is returned on the stack. (0 is returned if an error occurred.) On exit, the stack is as follows:



MLO-952-87

Error Returns

Two error codes can be returned in R0, with the carry bit set:

Code	Type	Description
ES$NMP	EX$RSC	Insufficient space for structure
ES$NLZ	EX$RSC	NEW of length zero

15.2.8 $DRPLY (Send Device Driver Reply)

The $DRPLY subroutine sends a standard device-driver reply packet to a reply semaphore, as specified in a packet that the caller of the routine supplies. If the supplied packet specifies a short reply (function modifier FM$BSM), the specified binary or counting semaphore is signaled. Otherwise, a full driver reply is sent to the specified queue semaphore.

If any error occurs, the packet is deallocated.

$DRPLY requires DRIVER or PRIVILEGED mapping, because it uses the SGLQ$ (Signal Queue Semaphore) primitive rather than the higher-level SEND$ primitive.

$DRPLY works in both mapped and unmapped systems.

The example in Section 15.1.3 illustrates the use of $DRPLY.

Calling Syntax

```
JSR   PC,$DRPLY
```

Input

R4 points to the packet to be returned. The 3-word reply semaphore field of that packet (offset DP.SEM) specifies the reply semaphore to be signaled, or 0 if none. The function word of that packet contains the FM$BSM modifier (bit 13), which must be set for short replies and cleared otherwise.

Output

R4 is cleared; all other registers are preserved.

15.2.9 $SV02, $SV03, and $SV05 (Save/Restore Registers)

$SV02, $SV03, and $SV05 are co-routines for saving and restoring registers. Each $SV0n routine saves registers 0 through n (where n=2, 3, or 5), calls the calling subroutine (JSR PC,@(SP)+), restores the registers, and returns. The net effect for the calling subroutine is to save registers in such a way that they are automatically restored upon exit from the calling subroutine. See the example below.

The $SV0n co-routines do not destroy the registers in the process of saving them. Thus, arguments can safely be passed in registers to a subroutine that calls an $SV0n co-routine.

The $SV0n routines work in both mapped and unmapped systems.

Calling Syntax

```
JSR   Rn,$SV0n
```

where n is 2, 3, or 5

Input

(Registers, stack, or none)

Output

After calling $SV0n, where n is 2, 3, or 5, n+2 words have been pushed on the stack. If you want to manipulate saved register m on the stack, the offset is (m*2)+2. See the example below.

Example

```
        JSR     PC,SUB              ; Call a subroutine
        ....                        ; Continue processing
        BR

SUB:    JSR     R2,$SV02            ; Save R0 - R2
        ....                        ; Perform some operation
        MOV     #error,<0*2>+2(SP); Put return value in saved R0
        RETURN                      ; Return, restoring registers
```

Appendix A
Directory Structure and File Storage

File-structured devices having a series of directory segments at the beginning of the device are called directory-structured devices. The directory segments contain entries describing the names, lengths, and creation dates of files on the device. Disks, diskettes, and TU58 cassettes may be directory-structured devices.

You can directly access any file on a directory-structured device, regardless of its location. Therefore, directory-structured devices are sometimes called random-access or block-replaceable devices.

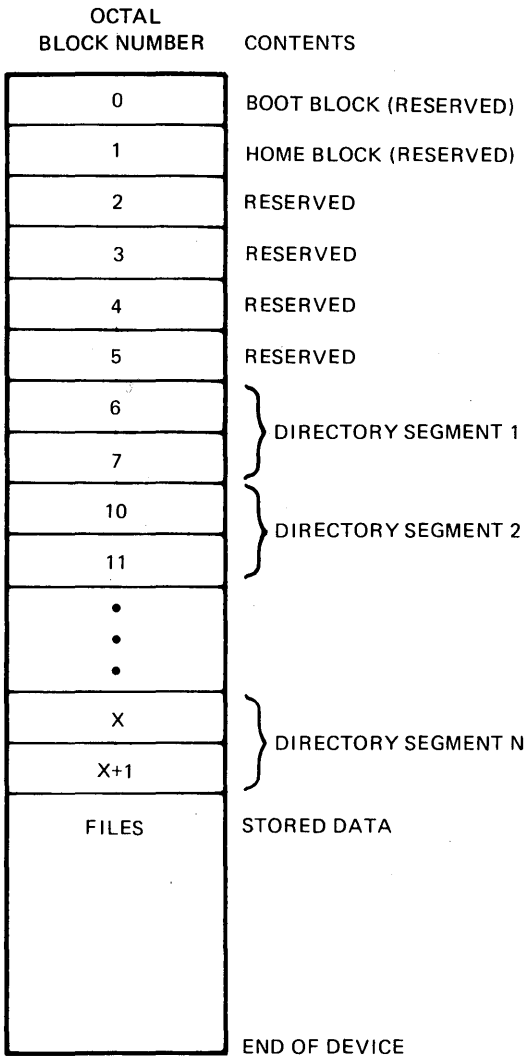
MicroPower/Pascal software includes an ancillary control process (ACP) and utility routines that can create and maintain directories. (Directory support must be enabled in the ACP prefix file; see Chapter 2.) The ACP stores files and maintains directories in the same format as the RT–11 file system.

This appendix first outlines the structure of a random-access device, then describes the contents of a device directory and explains how to split a directory segment, and lastly discusses file storage.

A.1 Structure of a Random-Access Device

A random-access device consists of a series of 256-word blocks. Blocks 0 to 5 are reserved for system use and cannot be used for data storage. The device directory begins at block 6. Figure A–1 shows the format of a random-access device.

Figure A–1: Format of Random-Access Device



A.1.1 Home Block

Block 1 of a random-access device is called the home block and contains information about the volume and its owner. Figure A–2 and Table A–1 show the format and contents, respectively, of the home block.

Figure A–2: Format of Home Block

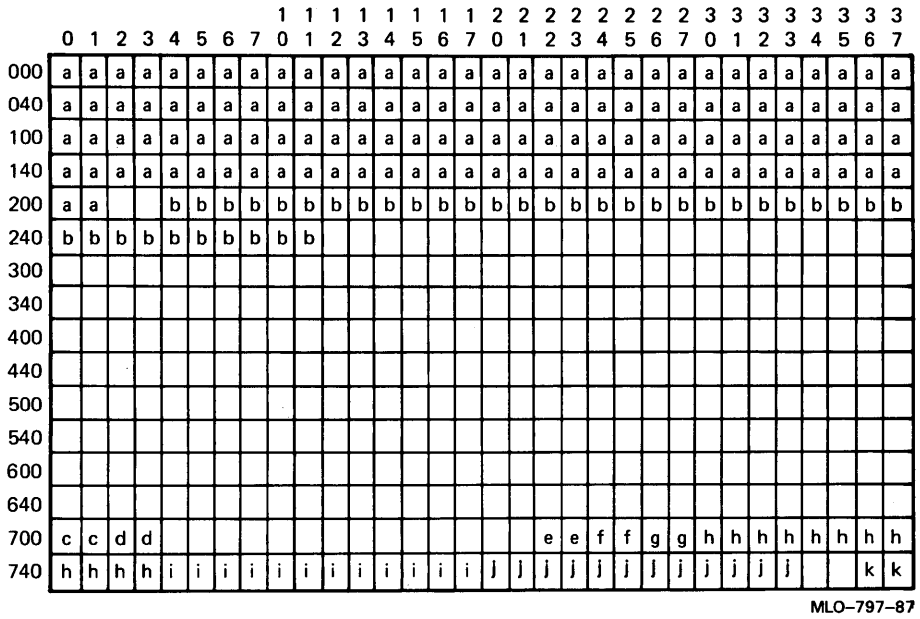


Table A–1: Contents of Home Block

Field	Location	Contents	Default
a	000–201	Bad-block replacement table	
b	204–251	INITIALIZE/RESTORE data area	
c	700–701	Radix–50 VRT (reserved for DIGITAL)	000000
d	702–703	Block number of first user file (reserved for DIGITAL)	000000
e	722–723	Pack-cluster size	000001
f	724–725	Block number of first directory segment	000006
g	726–727	System version	Radix–50 V3A
h	730–743	Volume identification	RT11A and 7 spaces
i	744–757	Owner name	12 spaces
j	760–773	System identification	DECRT11A and 4 spaces
k	776–777	Checksum	

In the home block, the contents of all areas except the checksum are undefined and are reserved for future use by DIGITAL. The checksum is computed by adding all the bytes into a word; then the word is negated.

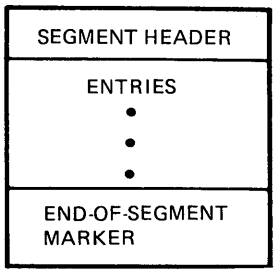
A.1.2 Directory

The directory of a random-access device consists of a series of two-block segments. Each segment is 512 words long and contains the names, lengths, and creation dates of files.

A directory can have from 1 to 31 segments. During device initialization, you establish the size of the directory area by determining the number of segments in the directory. In general, you should select many segments if you need to store many small files on a large device. To obtain more space for storing large files on a small device, you can select the minimal number of segments and reduce the size of the directory area.

Each directory segment consists of a 5-word segment header and entries containing file information. Each segment ends with an end-of-segment marker. Figure A–3 shows the general format of a directory segment.

Figure A–3: Format of Directory Segment



MLO–798–87

Note

An example program that reads a directory and prints the file names is included on the MicroPower/Pascal distribution kit. See the file FSPAS.PAS.

A.1.2.1 Directory Segment Header

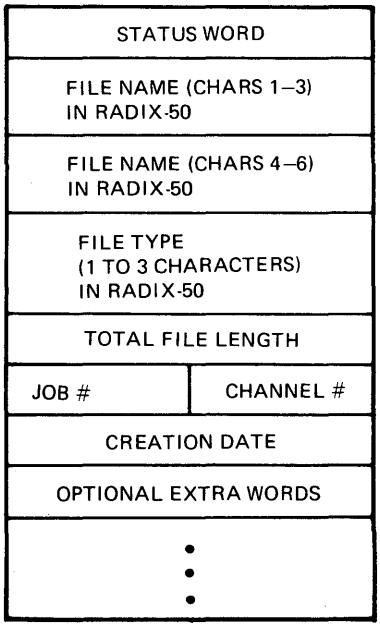
The directory segment header consists of five words; the remaining 507 words of the 2-block segment are for directory entries. The contents of the header words are as follows:

Word	Contents
1	The total number of segments in this directory. The valid range is from 1 to 31(decimal). If you do not specify the number of segments you require when you initialize the device, the default number of segments for that device is allocated.
2	The segment number of the next logical directory segment. The directory is a linked list of segments; word 2 is the link between the current segment and the next logical segment. If this word is 0, there are no more segments in the list.
3	The number of the highest segment in use. The ACP increments this counter each time it opens a new segment. Note that the system maintains this counter only in word 3 of the header for the first directory segment, ignoring the third word of the header of the other segments.
4	The number of extra bytes per directory entry; always an unsigned, even octal number; extended directory entries are described in Section A.1.2.3.
5	The block number on the device at which the stored data monitored by this segment begins.

A.1.2.2 Directory Entry

The rest of the directory segment consists of directory entries, followed by an end-of-segment marker. Figure A–4 shows the format of a directory entry.

Figure A–4: Format of Directory Entry



MLO–799–87

The first word of each directory entry is the status word, which describes the condition of the files stored on the device. The high-order byte of the status word contains a code representing the type of file (tentative, permanent, protected permanent). The low-order byte is reserved and should always be 0. Figure A–5 illustrates the status word.

Figure A–5: Format of Status Word

MLO–800–87

There are five valid status word values, as follows:

Status Word (octal)	Meaning
400	Tentative file.
1000	Empty area (the ACP does not use the name, file type, or date fields in an empty directory entry).
2000	Permanent file.
102000	Protected permanent file. (See Note below.)
4000	End-of-segment marker. (See Section A.1.2.4.)

The ACP uses three kinds of directory entries: tentative entries, empty entries, and permanent entries. These three entry types categorize areas as temporary data, available space on the device, or permanent data. The device directory always contains sufficient entries to describe the entire device.

- A tentative file is in the process of being created. When a process requests that a new file be created, the ACP creates a tentative file. The process must close the file to make the tentative file permanent. If you do not eventually close a tentative file, the system deletes it.
- An empty entry defines an area of the device that is available for use. Thus, when you delete a file, you create an empty area.
- A permanent file is a tentative file that has been closed. Permanent files are unique; that is, only one file can exist with a specific name and file type on a device. If another file exists with the same name and type when the program closes the current tentative file, the ACP deletes the first file, thus replacing the old file with the new file.

Note

The ACP provides a mechanism to prevent a file from being deleted. A file is protected when the high bit of its status word is set. Note that only permanent files can be protected. You can protect and unprotect files by using the RT-11 RENAME command, the IF$PRO and IF$UNP functions of the ACP (see Chapter 2), or the Pascal PROTECT_FILE and UNPROTECT_FILE routines (see Chapter 9 of the *MicroPower/Pascal Language Guide*).

The second, third, and fourth words in a directory entry contain the Radix-50 representation of the file name and file type. For empty areas, the ACP normally ignores these words.

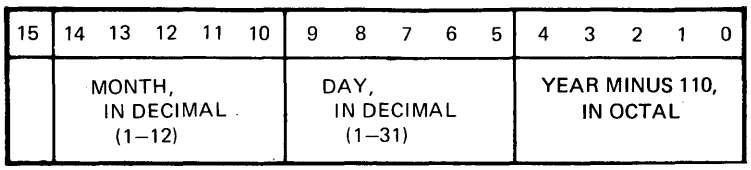
The fifth word in a directory entry contains the total file length—the number of blocks the file occupies on the device. Attempts to read or write outside the limits of the file result in an end-of-file error.

The sixth word in a directory entry contains a pointer to an active file in the ACP.

The seventh word of a directory entry contains the file's creation date. When a program creates a tentative file, the ACP moves the system date word into the creation date slot for the entry. To have your files dated, you must enable kernel clock services and call the SET_SYSTEM_DATE_TIME procedure. (See kit files DATTIM.PAS and TIMER.PAS.) If no

date was set, the date word equals 0. Figure A–6 shows the format of the date word. Bit 15 is used to indicate the end-of-file status.

Figure A–6: Format of Date Word



MLO–801–87

A.1.2.3 Extended Directory Entry

Normally, directory entries are seven words long, but by using the RT–11 DUP utility program with the /Z:n option, you can allocate extra words for each directory entry when you initialize the device. The fourth word of the directory segment header contains the number of extra bytes you specify. Although the RT–11 DUP command lets you allocate extra words, the ACP provides no means to manipulate this extra information conveniently. When the ACP initializes a directory, no extra bytes are allocated. Any program that needs to access these words must perform its own operations on the directory. Programs that manipulate the directory should use bit-test (BIT) instructions rather than compare (CMP) instructions.

A.1.2.4 End-of-Segment Marker

The ACP uses the end-of-segment marker, a status word of 4000 (octal), to determine when it has reached the end of the directory segment during a directory search. Note that an end-of-segment marker can appear as the last word of a segment. It does not have to be followed by a file name, file type, or other entry information.

A.2 Directory Use

A.2.1 Sample Directory Segment

The directory listing shown in Figure A–7 describes a double-density diskette with 11 files.

Figure A–7: Directory Listing

```
DIRECTORY/FULL DY0:

 29-APR-79
SWAP   .SYS      24       19-FEB-79      RT11SJ.SYS      65      19-FEB-79
< UNUSED >       77                      PIP    .SAV      16      19-FEB-79
DUP    .SAV      21       19-FEB-79      DIR    .SAV      17      19-FEB-79
EDIT   .SAV      19       19-FEB-79      LINK   .SAV      37      19-FEB-79
LIBR   .SAV      20       19-FEB-79      DUMP   .SAV       7      19-FEB-79
MACRO  .SAV      45       19-FEB-79      SIPP   .SAV      13      29-APR-79
< UNUSED >       613
 11 FILES, 284 BLOCKS
 690 FREE BLOCKS

DIRECTORY/SUMMARY DY0:

 29-APR-79

    11 FILES IN SEGMENT 1

     4 AVAILABLE SEGMENTS, 1 IN USE

 11 FILES, 284 BLOCKS
 690 FREE BLOCKS



>
```

MLO-1104-87

Figure A–8 shows the contents of segment 1 of the diskette directory, obtained by dumping absolute block number 6 of the device.

Figure A–8: Directory Segment

HEADER:	4	FOUR SEGMENTS AVAILABLE
	0	NO NEXT SEGMENT
	1	HIGHEST OPEN IS #1
	0	NO EXTRA BYTES PER ENTRY
	16	FILES START AT DEVICE BLOCK 16 OCTAL
ENTRIES:	2000	PERMANENT FILE
	75131	RADIX-50 FOR SWA
	62000	RADIX-50 FOR P
	75273	RADIX-50 FOR SYS
	30	FILE IS 30 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED ON 19-FEB-79
	2000	PERMANENT FILE
	71677	RADIX-50 FOR RT1
	142302	RADIX-50 FOR 1SJ
	75273	RADIX-50 FOR SYS
	101	101 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED ON 19-FEB-79
	1000	EMPTY AREA (THE FILE DXMNFB.SYS WAS DELETED)
	16315	RADIX-50 FOR DXM
	54162	RADIX-50 FOR NFB
	75273	RADIX-50 FOR SYS
	115	115 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	62570	RADIX-50 FOR PIP
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	20	20 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	16130	RADIX-50 FOR DUP
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	25	25 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	15172	RADIX-50 FOR DIR
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	21	21 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	17751	RADIX-50 FOR EDI
	76400	RADIX-50 FOR T
	73376	RADIX-50 FOR SAV
	23	23 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79

MLO–802–87

(Continued on next page)

Figure A–8 (Cont.): Directory Segment

2000	PERMANENT FILE
46166	RADIX-50 FOR LIN
42300	RADIX-50 FOR K
73376	RADIX-50 FOR SAV
45	45 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
46152	RADIX-50 FOR LIB
70200	RADIX-50 FOR R
73376	RADIX-50 FOR SAV
24	24 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
16125	RADIX-50 FOR DUM
62000	RADIX-50 FOR P
73376	RADIX-50 FOR SAV
7	7 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
50553	RADIX-50 FOR MAC
71330	RADIX-50 FOR RO
73376	RADIX-50 FOR SAV
55	55 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
74070	RADIX-50 FOR SIP
62000	RADIX-50 FOR P
73376	RADIX-50 FOR SAV
15	15 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
11647	CREATED 29-APR-79
1000	EMPTY AREA (NEVER USED SINCE INITIALIZATION)
000325	RADIX-50 FOR EM (STORED AT INITIALIZATION)
063471	RADIX-50 FOR PTY (STORED AT INITIALIZATION)
023364	RADIX-50 FOR FIL (STORED AT INITIALIZATION)
1145	**1145 OCTAL BLOCKS LONG**
–	USED ONLY FOR TENTATIVE FILES
–	(THE DATE IS NOT SIGNIFICANT)
4000	END-OF-SEGMENT-MARKER

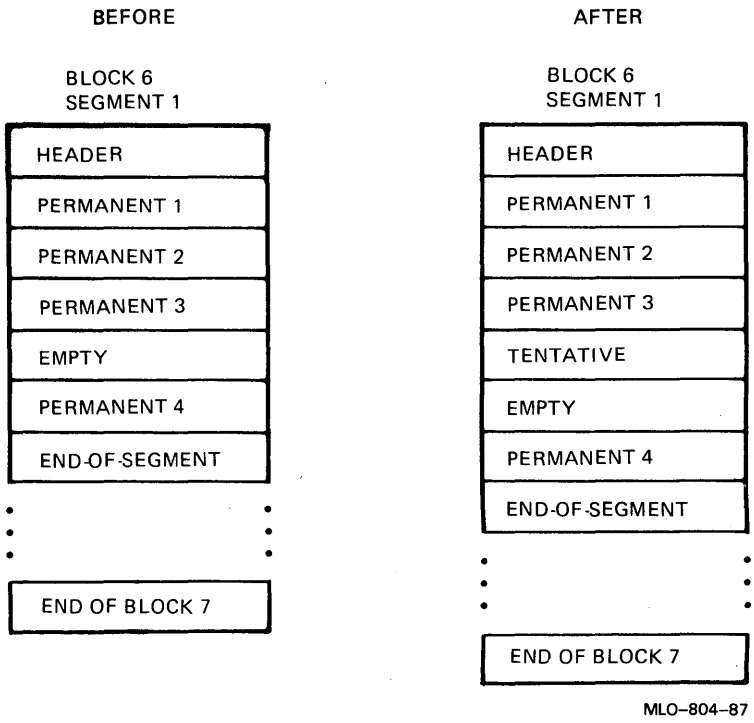
MLO-803-87

To find the starting block of a particular data file, first find the directory segment containing the directory entry for that file. Next, add to the starting block number (the fifth word of that directory segment header) the length of each permanent, tentative, and empty entry in the directory before your file. In Figure A–8, for example, the permanent file RT11SJ.SYS begins at block number 46 (octal)—starting block (16) plus SWAP.SYS length (30)—on the device.

A.2.2 Splitting a Directory Segment

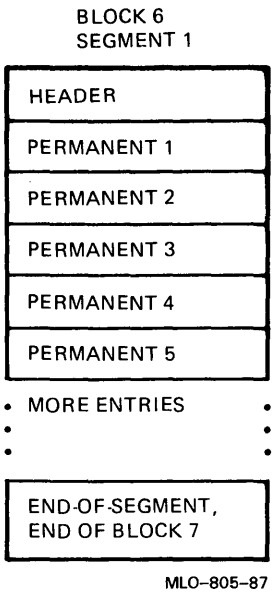
Whenever the ACP stores a new file on a volume, it searches through the directory for an empty area that is large enough to accommodate the new tentative file. When it finds a suitable empty area, it creates the new file as a tentative file followed by an empty area, sliding the rest of the directory entries down to make room for the new entry. Figure A–9 shows how the ACP stores a new file as a tentative file followed by an empty area.

Figure A–9: Storing a New File



This procedure works properly if the empty entry and the entries following it can move downward. If the segment is full, however, the ACP must split the segment, if possible, in order to store the new entry. Figure A–10 illustrates a directory segment that is full.

Figure A–10: Full Directory Segment



First, the ACP checks the header for the number of available segments. If there are none, a "directory full" error results, and you cannot store the new file. You can consolidate empty space on the volume at this point by using RT–11 commands or the Pascal SQUEEZE_DIRECTORY procedure (kit file SQUEEZ.PAS) to pack the directory segments and then can try the operation again. You could also delete any unneeded files.

If another directory segment is available, the ACP divides the current segment by first finding a permanent or tentative entry near the middle of the segment and saving its first word. In place of the first word, the ACP puts an end-of-segment marker. It then saves the current link information, links the current segment to the next available segment, and writes the current segment back to the volume.

Next, the ACP restores the first word of the middle entry to the copy of the segment that is still in memory and restores the link information. The ACP slides the middle entry and all subsequent entries to the top of the segment. Then the ACP writes this segment to the volume as the next available segment. Finally, the ACP reads segment 1 into memory and updates the information in its header, at which point the ACP begins its search again for a suitable empty entry to accommodate the new file.

Note

Throughout this process the Extra-Bytes word remains unchanged.

Figures A–11 and A–12 summarize the process of splitting a directory segment. In this example, segment 1 was the only segment in use. It contained an empty entry, but did not have room for a tentative entry in addition to the empty one. After the split, segments 1 and 2 are both about half full.

Figure A–11: Directory Before Splitting

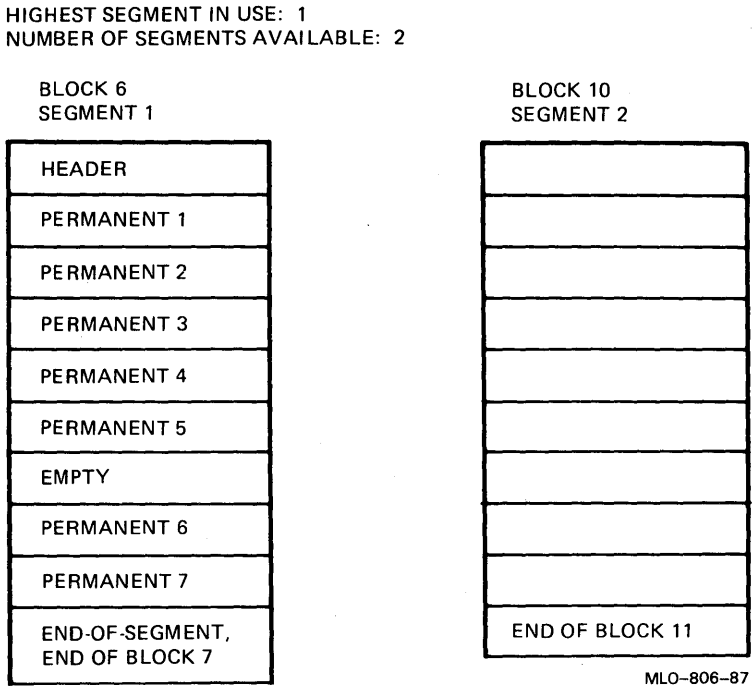
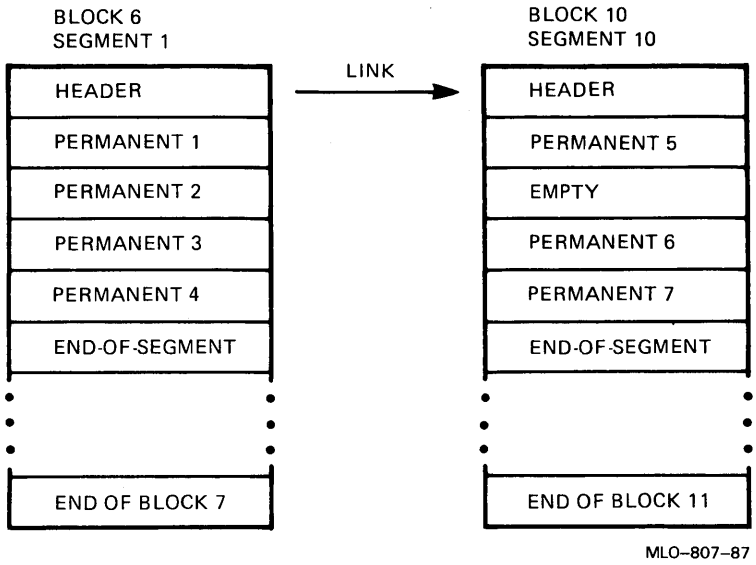


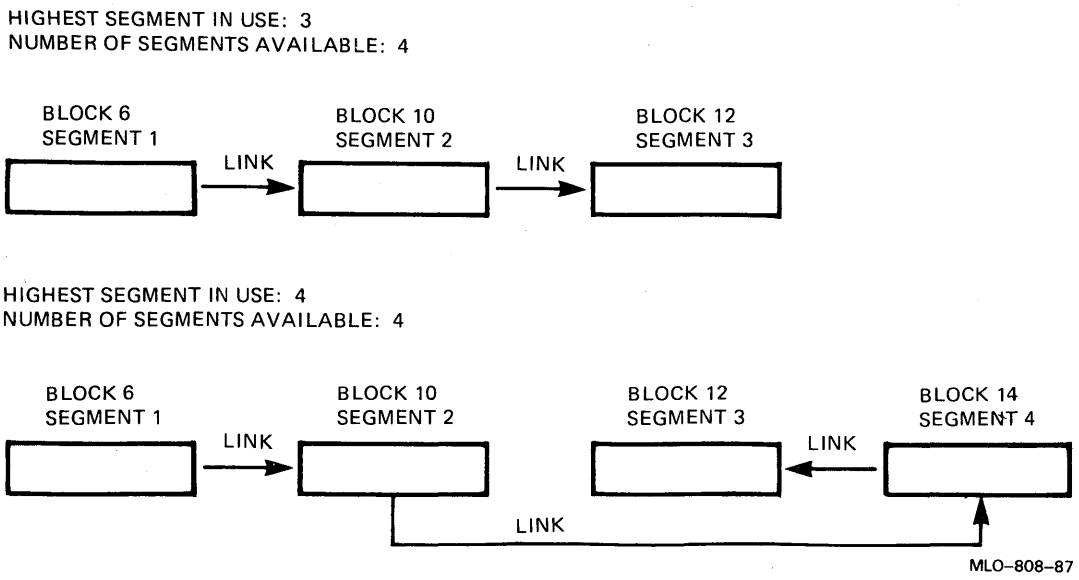
Figure A–12: Directory After Splitting



After a directory segment splits, the ACP can store the new file in either the new segment or the old one, depending on which segment now contains the empty area. Segment 2 contains the empty area in Figure A–12.

Thus far, the link word (word 2 of segment header) seems superfluous, since the segments are always in numerical order. However, consider a situation in which four segments are available: segment 1 fills and overflows into segment 2; segment 2 fills and overflows into segment 3; and segments 1, 2, and 3 are half full and are linked in the order in which they are located on the volume (blocks 6, 10, and 12). The picture changes if you delete a large file from segment 2, leaving a large empty entry, and add many small files to the volume. Segment 2 now fills up and overflows into the next free segment, segment 4, so that the links become visibly significant: segment 1 links to 2, segment 2 links to segment 4, and segment 4 links to segment 3 because segment 2 previously linked to segment 3. Figure A–13 illustrates this example.

Figure A–13: Directory Links



A.2.3 File Storage

The ACP uses the tentative, empty, and permanent entry types to describe completely the contents of a random-access device. All files reside on blocks that are contiguous on a device. There are several advantages and disadvantages to this method of storing data.

A.2.4 Method

When data is stored in contiguous blocks, I/O is more efficient. Transfers to large buffers are handled directly by the hardware for certain disks; seeks between blocks and program interrupts between blocks are eliminated. File data is processed simply and efficiently, since the data is not encumbered by link words in each block. Routines to maintain the directory are relatively small, because the directory structure is simple. File operations, such as open, delete, and close, are performed quickly, with few disk accesses, because only the directory must be accessed, not additional bit maps or retrieval pointers.

One disadvantage of this method of storing data is that a small device can become fragmented, requiring a squeeze operation to consolidate its free space. Another disadvantage is that once a file is closed, a running program cannot easily increase its size. Only a small number of output files can be opened simultaneously, even on a large device, unless the limits of the file sizes are known in advance. Finally, this scheme precludes the use of multiple and hierarchical directories.

In summary, any method of storing data has its advantages and disadvantages. The contiguous block method is used because its simple structure and low overhead suit typical MicroPower/Pascal applications.

Figure A–14 shows a simplified diagram of a random-access device that has a total of 250 blocks of space available for files after blocks 0 to 5 and the directory are accounted for. The device in the figure has two permanent files and one empty area stored on it.

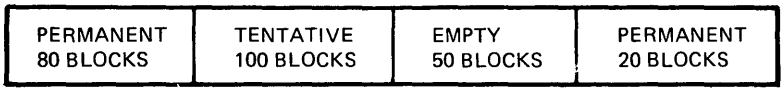
Figure A–14: Random-Access Device with Two Permanent Files

MLO–809–87

When you create a file, your program must allocate the space for the file. If you do not know the actual size, as is often the case, the space you allocate should be large enough to accommodate all the data possible.

The ACP creates a tentative file on the device with the length you specified. The tentative file must always be followed by an empty area to enable the ACP to recover unused space if less data is written to the file than you originally estimated. Figure A–15 shows a tentative file whose allocated size is 100 blocks. Note that the total amount of space on the device, 250 blocks in this case, remains constant.

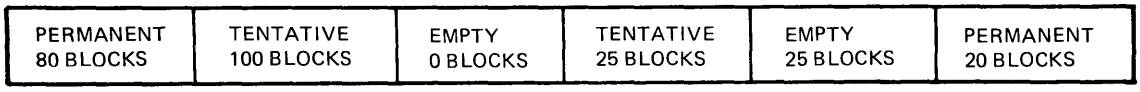
Figure A–15: Random-Access Device with One Tentative File



MLO–810–87

Suppose, for example, that while the file is being created by one process, another process enters a new file, allocating 25 blocks for it. The device would appear as shown in Figure A–16. Remember that every tentative file must be followed by an empty area.

Figure A–16: Random-Access Device with Two Tentative Files

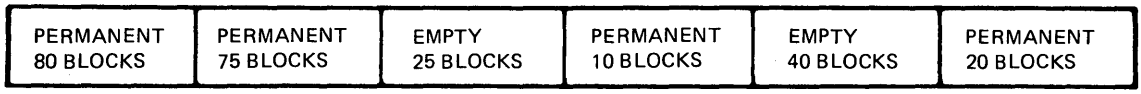


MLO–811–87

When the ACP finishes writing data to the device, it closes the tentative file. The ACP then makes the tentative file permanent. The length of the file is the actual size of the data that was written. The size of the empty area is its original size, plus any unused space from the tentative file.

Figure A–17 shows the same device after both tentative files are closed. The first file's actual length is 75 blocks; the second file's length is 10 blocks.

Figure A-17: Random-Access Device with Four Permanent Files



MLO-812-87

This method of storing files makes it impossible to extend the size of an existing file from within a running program. To make an existing file appear to be bigger, you can read the existing file, allocate a new, larger tentative file, and write both the old and new data to the new file. You can then delete the old file.

A.2.5 Size and Number of Files

The number of files you can store on a MicroPower/Pascal device depends on the number of segments in the device's directory and the number of extra words per entry. If you use no extra words, each segment can contain 72 entries.

The maximum number of directory segments on a MicroPower/Pascal device is 31 (decimal). Use the following formula to calculate the theoretical maximum number of directory entries and, thus, the maximum number of files:

$$31 * \frac{512 - 5}{7 + N} - 2$$

In the formula above, N represents the number of extra information words per directory entry. If N is 0, the maximum number of files you can store on the device is 2230 (decimal).

Note that all divisions are integer and that the remainder should be discarded.

In the formula above, the -2 is required for two reasons. First, in order to create a file, the tentative file must be followed by an empty area. Second, a 1-word end-of-segment entry must exist.

If you store files sequentially (that is, one immediately after another) without deleting any files, roughly one-half the theoretical maximum number of files will fit on the device before a directory overflow occurs. This situation results from the way the ACP splits filled directory segments.

When a directory segment becomes full, requiring the opening of a new segment, the ACP moves approximately one-half of the directory entries of the filled segment to the new segment. Thus, when the final segment is full, all previous segments have approximately one-half of their total capacity. See Section A.2.2 for a detailed explanation of how the ACP splits a directory segment.

If you add files continually to a device without issuing the RT-11 SQUEEZE monitor command or calling the Pascal SQUEEZE_DIRECTORY procedure (see kit file SQUEEZ.PAS), you can use the following formula to compute the maximum number of entries and, thus, the maximum number of files:

$$(M - 1) * \frac{S}{2} + S$$

In the formula above, M represents the maximum number of segments.

You can use the following formula to compute S:

$$S = \frac{512 - 5}{7 + N} - 2$$

In the formula above, N represents the number of extra information words per entry.

You can realize the theoretical total of directory entries (see the first formula above) by using the RT–11 SQUEEZE command or the Pascal SQUEEZE_DIRECTORY procedure to compress the device when the directory fills up.

Appendix B

KXT11-CA and KXJ11-CA Peripheral Processors

This appendix contains information you will need for writing MicroPower/Pascal applications for the KXT11-CA or KXJ11-CA Peripheral Processor. Additional information for KXT11-CA/KXJ11-CA device drivers is contained in the following chapters:

Chapter	Driver
3	Asynchronous serial line/terminal (TT)
4	TU58 (DD)
6	KXT11-CA/KXJ11-CA parallel lines and timer/counters (YK)
9	KXT11-CA/KXJ11-CA DMA transfer controller (QD)
13	KXT11-CA/KXJ11-CA two-port RAM (KX and KK), KXT11-CA/KXJ11-CA synchronous serial line (XS)

B.1 KXT11-CA/KXJ11-CA Hardware and Applications

The KXT11-CA/KXJ11-CA Peripheral Processor is an LSI-11, single-board, 16-bit computer with local memory and communication ports. You can use it as a self-contained stand-alone system or as a component (peripheral processor) of an LSI-11-based multiple processor system.

In a multiple processor system, you can add up to 14 user-programmed KXT11-CA/KXJ11-CA Peripheral Processors to traditional Q-bus systems and communicate with them from the LSI-11 CPU acting as arbiter. The software architecture is master/slave (not to be confused with the bus-master/bus-slave hardware concept) which means the KXT11-CA/KXJ11-CA application (slave) performs operations only on command from the arbiter application (master). The master application runs in the Q-bus arbiter processor and controls the KXT11-CA/KXJ11-CA application by sending it messages over the KXT11-CA/KXJ11-CA two-port RAM (TPR) registers in the I/O page. The KXT11-CA/KXJ11-CA can also transfer data to and from main memory with its DMA transfer controller (DTC) facility.

When configured for stand-alone operation, the KXT11-CA or KXJ11-CA is completely self-contained with no Q-bus required.

MicroPower/Pascal supports KXT11–CA and KXJ11–CA single-board computers as stand-alone systems or as peripheral processors in a multiple processor environment. You can also program the KXT11–CA/KXJ11–CA using the MACRO–11 language. In peripheral processing applications, you can then incorporate the processors into arbiter applications based on the RT–11, RSX–11M, RSX–11M–PLUS, Micro/RSX, MicroVMS, or MicroPower/Pascal operating environment.

In addition to DIGITAL's standard Q-bus software development tools, you can choose from the following five software products that provide tools for KXT11–CA/KXJ11–CA application development:

- MicroPower/Pascal
- Peripheral Processor Tool Kit-RT–11
- Peripheral Processor Tool Kit-RSX
- Peripheral Processor Tool Kit-Micro/RSX
- Peripheral Processor Tool Kit-MicroVMS

MicroPower/Pascal software provides tools for developing KXT11–CA/KXJ11–CA stand-alone or peripheral processor applications in MicroPower/Pascal and MACRO–11 under the control of the MicroPower/Pascal run-time kernel. Included are drivers for the following KXT11–CA/KXJ11–CA on-board devices:

- Asynchronous serial I/O
- Synchronous serial I/O
- Parallel I/O and counter-timers
- Real-time clock
- DMA transfer

For peripheral processor applications, device drivers provide communication through the TPR. They allow a MicroPower/Pascal application on the KXT11–CA/KXJ11–CA to communicate with a MicroPower/Pascal, RT–11, RSX–11, or MicroVMS application in the arbiter processor. MicroPower/Pascal provides a device driver for MicroPower/Pascal arbiter applications. The drivers for RT–11, RSX–11, and MicroVMS are available in the tool kits.

KXT11–CA/KXJ11–CA peripheral processor tool kits provide tools for using peripheral processors in traditional Q-bus systems. Support is provided for RT–11, RSX–11M, RSX–11M–PLUS, Micro/RSX, and MicroVMS arbiter applications to load and communicate with KXT11–CA/KXJ11–CA peripheral processors across the Q-bus. The drivers communicate with KXT11–CA/KXJ11–CA processors programmed in MicroPower/Pascal.

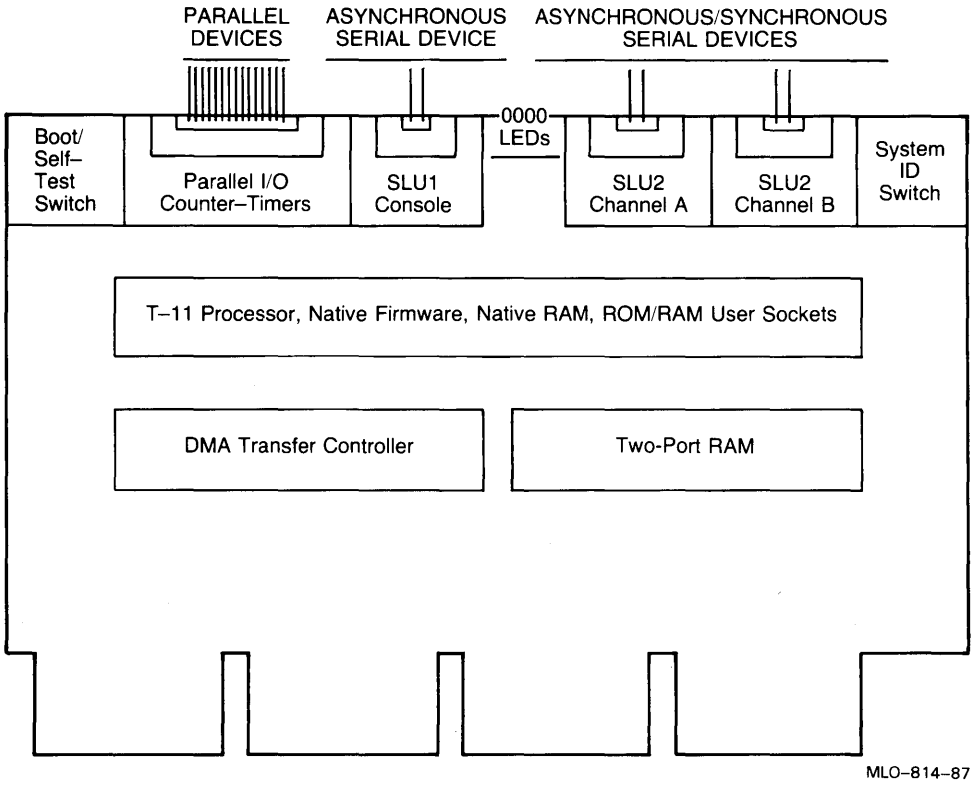
If those tools do not meet your needs, you can program the KXT11–CA or KXJ11–CA in the MACRO–11 assembly language, using the standard PDP–11 application development tools (MACRO–11, LINK, ODT, MACDBG, and so on).

If your KXT11–CA/KXJ11–CA application program uses ROM, you can load it with the DECprom program (VMS systems).

B.1.1 KXT11-CA Hardware Features

Figure B-1 shows the general layout of the following major hardware components. DIGITAL-supplied software supports most but not all hardware features; the software documentation describes the supported features.

Figure B-1: KXT11-CA Hardware Features



- DIGITAL DCT-11 microprocessor—A 16-bit, 7-MHz microprocessor that executes the PDP-11 basic instruction set.
- On-board memory—Local memory consisting of 32K bytes of static read/write memory (RAM), sockets for up to 32K bytes of PROM or static RAM, and 8K bytes of native firmware. Additional features include eight memory map configurations and battery backup for native RAM.
- Native firmware—Provides:

  Power-up self-test

  Optional loopback tests

  Hardware initialization

  Serial ODT accessible from a console terminal or via the Q-bus

Application start-up: ROM, TU58 boot, or load from the Q-bus arbiter

- TPR—A 16-word interface to the Q-bus that passes control information and data between the KXT11–CA and the arbiter. The RAM is divided into three areas: one area for KXT11–CA native firmware commands and two areas for application message passing.
- 2-channel DTC—A device that provides for memory and I/O data transfers between the local KXT11–CA and global memory on the Q-bus, using direct memory access. Permitted transfer combinations are:

  Local memory to global memory

  Global memory to local memory

  Local memory to local memory

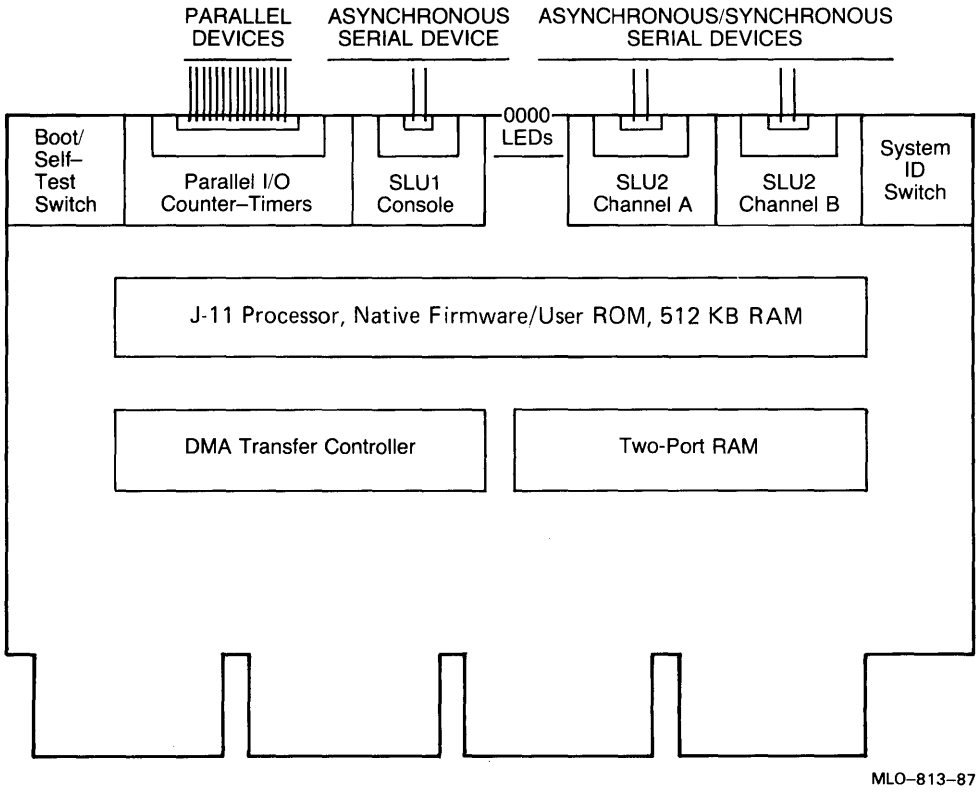
  Global memory to global memory
- Parallel I/O interface—Contains two bidirectional 8-bit input/output ports, one 4-bit control port, and three 16-bit programmable counter-timers.
- Console port (DLART)—Provides DL11-compatible asynchronous serial communication.
- Multiprotocol controller—Provides 2-channel synchronous/ asynchronous serial I/O communication.
- System ID switch—Sets the system identification address and establishes the KXT11–CA for Q-bus or stand-alone operation.
- Boot/Self-test switch—Selects bootstrap and firmware self-test operations.
- Configuration jumpers—Electrical jumpers on the KXT11–CA circuit board that select some of the hardware configuration options (other options are software-configurable).

See the *KXT11–CA Single-Board Computer User's Guide* for detailed information about the KXT11–CA hardware.

B.1.2 KXJ11–CA Hardware Features

Figure B–2 shows the general layout of the following major hardware components. DIGITAL-supplied software supports most but not all hardware features; the software documentation describes the supported features.

Figure B–2: KXJ11–CA Hardware Features



- J–11 (DCJ11–AC) 16-bit microprocessor

  Executes extended PDP–11 instruction set (140 instructions including floating-point).

  Contains memory management unit for three levels of memory protection and 4M byte addressing.

  Operates at 14 MHz.

- Memory

  512K bytes of dynamic RAM

  Can be accessed by local (on-board) devices and, if enabled as shared memory, also by Q-bus devices

  Up to 64K bytes of PROM; 16K bytes of which is for firmware

- Native firmware—Provides:

  Power-up self test

  Optional loopback tests

Hardware initialization

Application start-up: ROM, TU58 boot, or load from the Q-bus arbiter

- TPR—A 16-word interface to the Q-bus that passes control information and data between the KXJ11–CA and the arbiter. The RAM is divided into three areas: one area for KXJ11–CA native firmware commands and two areas for application message passing.
- 2-channel DTC—A device that provides for memory and I/O data transfers between the local KXJ11–CA and global memory on the Q-bus, using direct memory access. Permitted transfer combinations are:

  Local memory to global memory

  Global memory to local memory

  Local memory to local memory

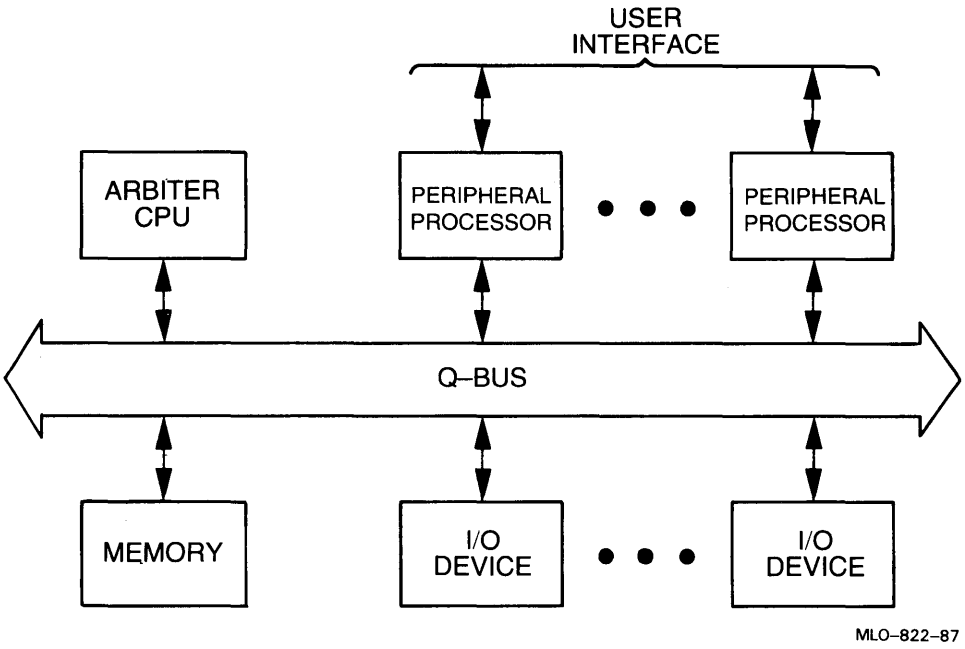
  Global memory to global memory
- Parallel I/O interface—Contains two bidirectional 8-bit input/output ports, one 4-bit control port, and three 16-bit programmable counter-timers.
- Console port (DLART)—Provides DL11-compatible asynchronous serial communication.
- Multiprotocol controller—Provides 2-channel synchronous/asynchronous serial I/O communication.
- System ID switch—Sets the system identification address and establishes the KXJ11–CA for Q-bus or stand-alone operation.
- Boot/Self-test switch—Selects bootstrap and firmware self-test operations.
- Configuration jumpers—Electrical jumpers on the KXJ11–CA circuit board that select some of the hardware configuration options (other options are software-configurable).

See the *KXJ11–CA Single-Board Computer User's Guide* for detailed information about the KXJ11–CA hardware.

B.1.3 Using the KXT11–CA or KXJ11–CA as a Peripheral Processor

Peripheral processor applications help you off-load tasks from a conventional LSI–11 processor application. This improves overall system performance by distributing the application task load. You can add up to 14 peripheral processors (KXTs and/or KXJs) to a traditional LSI–11 (Q-bus) system configuration in the same way you add other I/O device controllers. (See Figure B–3.) The difference between the KXT or KXJ and conventional I/O devices is that the KXT and KXJ are user programmable while conventional I/O devices are not.

Figure B–3: Adding Peripheral Processors to Traditional LSI–11 Systems



- Guaranteed response time—A dedicated KXT or KXJ can ensure a specific interrupt response time.
- Data collection and reduction—The arbiter is relieved of the overhead of the interrupts required to control devices and the CPU Q-bus time required to refine and format the data.
- Machine control—The arbiter processor gives high-level commands to the peripheral processor. The peripheral processor translates the commands to the level required by the device(s) and monitors progress. The arbiter application merely issues commands and manages higher-level application tasks.
- Communication protocol handling—The peripheral processor relieves the arbiter processor from the tasks of handling communication line interrupts, packing/unpacking messages, and formatting them. Only data is transferred to and from the arbiter.

B.1.3.1 Peripheral Processor Hardware Configuration

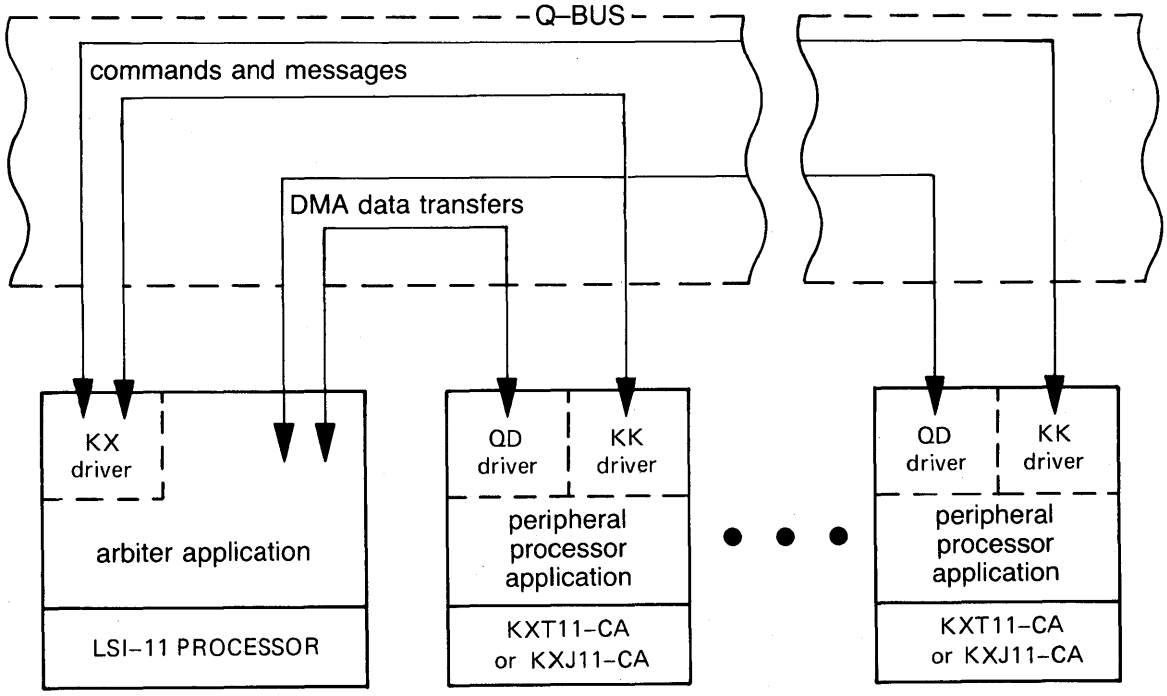
The application system consists of one Q-bus arbiter CPU (for example, LSI–11/23, LSI–11/73, or SBC–11/21–PLUS) and up to 14 peripheral processors attached to the Q-bus. The KXT and KXJ cannot be bus arbiters.

Communication between the arbiter and the peripheral processor takes place over the Q-bus through the TPR. The TPR's control, status and data registers, and vectors appear in the arbiter's I/O page. Programs can access the registers in ways similar to those of I/O devices. The peripheral processor's hardware configuration options determine where its TPR area appears in the arbiter's I/O page.

B.1.3.2 Peripheral Processor Application Software Configuration

A master application in the arbiter processor directs slave peripheral processor operations (Figure B–4). Communication takes place over the Q-bus, using messages to transfer data and to control the KXT or KXJ hardware and its application software. Communication can take place using the TPR or the DTC. If desired, the peripheral processor can interrupt the arbiter application when it completes the requested task.

Figure B–4: Peripheral Processor Application Software Configuration



B.2 Developing KXT11-CA and KXJ11-CA Applications

This section describes both the procedures for developing a peripheral processor application and the specific considerations for such applications.

B.2.1 Partitioning the Application

The first step in developing applications using one or more KXT11-CAs or KXJ11-CAs is to determine if your application can be partitioned to take advantage of multiple processors. There must be a set of processes that can be performed usefully in parallel and within the capabilities of the KXT or KXJ hardware. In addition, it must be possible to direct and monitor the progress of the process through messages or transfer of blocks of data. Some characteristics of processes that are good candidates for KXT or KXJ applications are:

- Input/Output processes with critical interrupt latency requirements—By assigning processes with critical interrupt latency requirements to dedicated peripheral processors, you can ensure that the rest of the application does not interfere with the service of critical devices.
- Input/Output processes with a high frequency of interrupts—peripheral processors can relieve the arbiter from the continual context switching required to process interrupt-driven I/O.
- Input/Output data reduction processes—By assigning one or more peripheral processors to I/O processes that require large quantities of input data and produce a small amount of output, you can save arbiter processing time. The peripheral processor receives the data, decodes it, and reduces it to the required subset, discarding the rest.
- Computational processes—The KXT or KXJ can perform parallel computational operations by using the DTC to transfer data directly to its memory, perform the operation, and transfer the data back to Q-bus memory. On a KXJ, if the data is in KXJ memory, it can be accessed directly from the arbiter and the KXJ. There is then no need to transfer the data from Q-bus memory to local memory and back.
- Real-time control functions—You can assign a KXT or KXJ to control functions that require constant interaction with a device but little interaction with the main application. The arbiter can then direct the peripheral processor with high-level commands.

B.2.2 Designing the Peripheral Processor Application System

For a peripheral processor application, you must design an application-level communication protocol between the arbiter application and the peripheral processor to command the peripheral processor (conceptually an intelligent I/O device) to perform its functions—for example, start, stop, and transfer data. The commands are generally formatted into messages and sent through the 2-port RAM (TPR). MicroPower/Pascal provides the KX/KK device driver pair to facilitate TPR communication.

In addition to commands, data can be sent to the peripheral processor either by using the TPR to send data as messages or directly by using the DMA transfer controller (DTC). (The choice of method is governed by such criteria as the amount of data and the frequency of transmission.)

When you use the DTC, the arbiter typically passes to the peripheral processor a message specifying the location (Q-bus physical address) and size of the data buffer to transfer. The peripheral processor application then directs the DTC locally to make the transfer.

In general, you should use the TPR to send small or infrequently issued messages. You should use the DTC to send large or frequently issued messages, especially if it can be done in parallel with other peripheral processor processes. When to use the DTC depends on the application and must be determined on a case-by-case basis.

B.2.3 Software and Hardware Configuration Guidelines

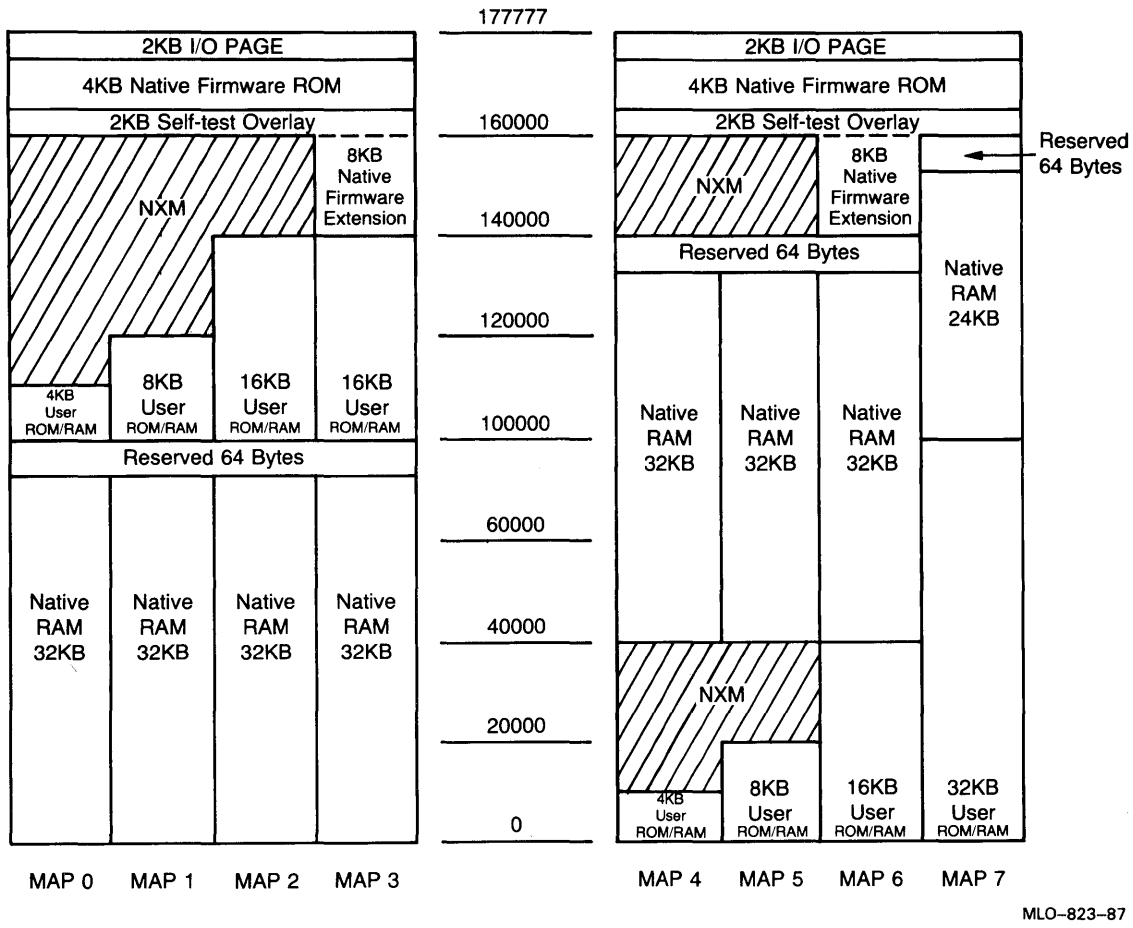
This section tells you how to configure the MicroPower/Pascal software and KXT11–CA and KXJ11–CA hardware. There are four main areas of concern:

- The location of RAM and ROM
- The system configuration, stand-alone or peripheral processor
- The I/O device options
- The bootstrap start-up option

B.2.3.1 Configuring Memory

KXT11–CA native memory has eight jumper-selectable configurations (maps), as shown in Figure B–5. Each map defines a particular combination of native (local) RAM and user-installed RAM or ROM residing in the user sockets of the KXT11–CA.

Figure B–5: KXT11–CA Memory Map Configurations



B.2.3.2 Memory Configuration Steps

When you configure KXT11–CA memory, you must specify parameters in the software configuration file and install jumpers on the KXT11–CA circuit board:

1. Select a map according to the requirements of your application.
2. Specify the number of the map you selected in the KXT11C configuration file macro.
3. Install additional ROM or RAM in the user sockets.

Note

If your application uses ROM, consider substituting RAM in its place during debugging so you can use the PASDBG debugging program.

4. In the configuration file's MEMORY macro, specify each contiguous block of RAM and ROM defined by the selected map and the ROM or RAM installed in the user sockets.
5. Install the map selection jumpers on the KXT11-CA circuit board. (The *KXT11-CA Single-Board Computer User's Guide* shows the location of these jumpers.)
6. Select an appropriate setting for the boot/self-test switch. (See Table B-2.)

B.2.3.3 Memory Selection Rules

MicroPower/Pascal software can use any KXT11-CA memory map option as long as you observe the following configuration rules:

1. You can select any memory map for RAM-only applications that is appropriate for the type of RAM devices installed in the user sockets.
2. You can select maps 4, 5, 6, or 7 for MicroPower/Pascal applications that use ROM. MicroPower/Pascal memory allocation rules require that you configure all ROM in memory addresses lower than those assigned to RAM. Thus, you should not specify maps 0 to 3 if your MicroPower/Pascal application uses ROM.
3. If your application will be loaded from TU58 DECtape II, you must configure RAM to start at address 0.
4. Do not configure your application to use the highest 64 bytes of native RAM. This area is reserved for KXT11-CA native firmware. The location of this 64-byte area depends on which memory map you select with the boot/self-test switch. (See Figure B-5.) If you are going to use RAM in the user sockets, assign it to low memory addresses (maps 4 to 7).
5. Do not configure your application to use nonexistent memory addresses (address space identified as NXM in Figure B-5).
6. Never configure the 8K-byte block of memory addresses shown in maps 3 and 6 and designated 8K-byte Native Firmware Extension. These addresses reference locations in the native firmware socket that are outside the address space of the native firmware ROM provided by DIGITAL.

Table B-1 summarizes MicroPower/Pascal map usage.

Table B-1: MicroPower/Pascal Usage of KXT11-CA Memory Maps

Map	Usage
0	32K bytes of native RAM; 4K bytes ROM/RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
1	32K bytes of native RAM; 8K bytes ROM/RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
2	32K bytes of native RAM; 16K bytes ROM/RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
3	32K bytes of native RAM; 16K bytes ROM/RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.

Table B–1 (Cont.): MicroPower/Pascal Usage of KXT11–CA Memory Maps

Map	Usage
4	32K bytes of native RAM; 4K bytes ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
5	32K bytes of native RAM; 8K bytes ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
6	32K bytes of native RAM; 16K bytes ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
7	24K bytes of native RAM; 32K bytes ROM/RAM in user sockets mapped low. Do not allocate memory between 157700 and 157777.

B.2.4 Configuring the KXT11–CA or KXJ11–CA System Environment

This section tells you how to set up the KXT11–CA or KXJ11–CA for its system environment. The features you can select include stand-alone operation or peripheral processor operation and the system's start-up options (application bootstrap device, console ODT operation, and self-test program execution). When you select these features, you must configure jumpers and switches on the KXT11–CA or KXJ11–CA circuit board and change some of the parameters of the KXT11C or KXJ11C macro in the configuration file.

The system ID switch and the boot/self-test switch specify to the native firmware the desired environmental and operational features of the KXT11–CA or KXJ11–CA. The boot/self-test switch determines when the self-tests will be performed and how the application program will be initialized.

B.2.4.1 Selecting Stand-Alone or Peripheral Processor Operation

You can select stand-alone or peripheral processor operation with the system ID switch on the KXT11–CA circuit board.

To use stand-alone mode, select switch position 0 or 1. In stand-alone mode, the TPR is disabled, since no Q-bus is required. To use peripheral processing mode, select switch positions 2 to 15. You can configure a maximum of 14 peripheral processors (KXT11–CAs and/or KXJ11–CAs) in a Q-bus system by selecting switch positions 2 to 15.

In peripheral processing mode, the TPR is enabled and connected to the Q-bus. Each switch position selects a different base address for the TPR registers in the arbiter's I/O page. The switch selects addresses from a high or low range, depending on whether or not you install the TPR base address jumper on the KXT11–CA or KXJ11–CA circuit board.

If you are using a KXJ11–CA as a peripheral processor for a board later than etch revision F1, configure the board so that the J–11 processor does not request a Q-bus grant for the bus lock instructions (TSTSET, WRTLCK, and ASRB). This action disables both the Q-bus lock capability and DMA bus timeouts for those instructions.

Section B.5 lists system ID switch settings and the associated I/O page base addresses of the TPR. When selecting the TPR base addresses, avoid conflicts between the KXT11–CA or KXJ11–CA processor's TPR I/O page registers and I/O page registers allocated to other devices on the application system's Q-bus.

Each system ID switch number you select must be unique among all system ID switch numbers for KXT11–CA or KXJ11–CA processors on the same Q-bus. The number need not be in a continuous sequence with the system ID switch numbers selected for other KXT11–CA or KXJ11–CA processors on the Q-bus.

Specify the CSR address implied by the system ID switch and TPR base address jumper setting you select in the KX device driver prefix file. You can insert this information in the file manually with an editor or automatically during execution of MPBUILD, as described in the *MicroPower/Pascal System User's Guide.*

B.2.4.2 Selecting KXT11–CA or KXJ11–CA Initialization and Self-Test Options

KXT11–CA and KXJ11–CA firmware provide initialization and diagnostic self-test functions that are selected by the boot/self-test switch on the KXT11–CA and KXJ11–CA circuit boards. They are power-up features that provide for hardware initialization, automatic self-tests, console ODT, application bootstrapping, and execution of control routines that handle local restart interrupts to allow the arbiter to gain control of the KXT11–CA or KXJ11–CA. Tables B–2 and B–3 summarize the boot/self-test switch functions for the KXT and KXJ respectively.

Table B–2: Initialization/Self-Test Options for the KXT11–CA

Initialize/Test Feature	Boot/Self-Test Switch Position*															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Start-up in ROM	X	X	X													
Boot from TU58 DECtape				X												
Load RAM from arbiter and run						X	X									
Enter debugging (ODT) mode					X											
Perform user ROM tests			X							X						
Perform auto self-tests	X	X	X		X					X						
Dedicated test mode									X	X	X					
Reserved								X				X	X	X	X	X

*Switch positions 7 and 11 to 15 are reserved. If you apply power to the KXT11-CA with these positions selected, it will not function and the LED display will indicate a fatal error. Do not use switch positions 8, 9, or 10 when using MicroPower/Pascal. These positions are for dedicated testing.

MLO-829-87

Table B-3: Initialization/Self-Test Options for the KXJ11-CA

Initialize/Test Feature	Boot/Self-Test Switch Position*															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Start-up in ROM	X	X	X						X	X	X					
Boot from TU58 DECtape				X								X				
Load RAM from arbiter and run						X	X							X	X	
Enter debugging (ODT) mode					X								X			
Perform user ROM tests			X								X					
Perform auto self-tests	X	X	X		X				X	X	X		X			
Dedicated test mode								X								X
Reserved																

*Do not use switch positions 7 through 15 when using MicroPower/Pascal.

MLO-828-87

ROM Application Start-up—Boot/Self-test switch positions 0, 1, and 2 allow you to execute a ROM application. The native firmware transfers control to the ROM code by emulating a trap to location 24. Consequently, you must configure the ROM to start at address 0 (maps 4 to 7) to ensure that the contents of vector 24 are preserved.

Switch position 0 inhibits the automatic self-tests. By using this switch position, you can reduce application start-up time to a minimum. Choose this position only when necessary to maintain acceptable application performance.

Switch position 1 inhibits the user ROM checksum test. This allows you to start an application that was loaded into ROM that contains no checksum or that was loaded into ROM with a checksum that was calculated by an algorithm that is incompatible with the test. Section B.9 describes the procedure to use with the DECprom program to calculate and program ROM checksums for KXT11-CA and KXJ11-CA applications. The *KXT11-CA Single-Board Computer User's Guide* and the *KXJ11-CA Single-Board Computer User's Guide* describe the checksum algorithms for the two boards.

Use switch position 2 if your ROM application contains a user ROM checksum.

TU58 and RSP Bootstrap—If you are going to load your application image from a TU58 DECtape II drive or an RSP (radial serial protocol) link, select switch position 3. This causes the TU58 primary bootstrap to execute on power-up and load your application using RSP from a TU58 DECtape II subsystem or other system through SLU1 (DLART). Once the primary bootstrap begins receiving the boot block from the TU58, it checks that the first word of the block is in the range 240 to 277 (octal). If true, it considers this block to be a valid boot block, loads the remaining blocks into RAM, and starts the program. If the first word's value is invalid, the primary bootstrap continues to check, alternating between unit 0 and unit 1, until it finds a valid boot block.

When your application will be loaded from TU58 DECtape II, you must configure RAM to start at address 0.

Loading from the Arbiter—You can load your peripheral processor application from a system storage device via the arbiter. If your arbiter application runs under MicroPower/Pascal, you can use the Pascal procedure KXT_LOAD or KXJ_LOAD described in Section B.10. If your arbiter application runs under RSX-11 or RT-11, you can use the KUI utility program described in the *Peripheral Processor Tool Kit/RSX Reference Manual* or *Peripheral Processor Tool Kit/RT Reference Manual.*

Boot/Self-test switch positions 5 and 6 instruct the KXT11-CA or KXJ11-CA to wait for a command over the Q-bus either from KXT_LOAD or KXJ_LOAD or from KUI. The automatic self-tests are performed first if you select switch position 5; they are inhibited if you select switch position 6.

Automatic Self-Tests—The automatic self-tests are a subset of the self-test functions that the native firmware provides. These tests are run when the boot/self-test switch is in positions 1, 2, 3, and 5.

The automatic self-tests include a:

- CPU test
- RAM test
- ROM test, native and user (if selected)
- CSR test, NXM test of all native CSRs and the TPR (read-only test)
- DMA test, local-to-local DMA transfers
- BEVENT test (KXJ11-CA only)

The tests report diagnostic errors, using the LED display on the KXT11-CA or KXJ11-CA circuit board and the TPR system control registers. The LED display reports automatic self-test diagnostic data and general KXT11-CA or KXJ11-CA status. Control registers 2 and 3 of the TPR contain the test status information on completion of the nonfatal tests. Register 2 indicates which tests failed, and register 3 specifies the type of failure. Register 3 is overwritten only if an error was encountered and will contain the valid discrete error code for the last test that found an error.

The native firmware considers failure of the following self-tests as a fatal condition and will not allow the application to run:

- CPU test
- Native RAM test (KXT11–CA only)
- Native ROM test
- CSR test (TPR portion)

The tests report fatal errors only in the LED indicators, since the firmware disables the TPR under these conditions.

The results of the remaining nonfatal tests do not prevent the application from running and are reported in the LED display and the TPR system control registers. The user application should check for these nonfatal error conditions to see if they affect the application.

The *KXT11–CA Single-Board Computer User's Guide* and the *KXJ11–CA Single-Board Computer User's Guide* discuss the meaning of all LED displays and the TPR registers and associated error codes.

If you do not select the automatic self-tests, you can reduce application start-up time to a minimum. However, these tests are useful diagnostic tools. You should bypass them only when necessary to maintain acceptable application performance.

Debugging (ODT) Mode—When you use boot/self-test switch position 4, the KXT11–CA enters console ODT through SLU1 (DLART). Select this switch position when you want to debug your application with PASDBG.

Dedicated Test Mode—This mode is for dedicated diagnostic testing; the tests permit no execution of application code. On a KXT11–CA you select the tests with boot/self-test switch positions 8, 9, and 10. Use of these switch positions causes RAM to be mapped to low memory addresses by overriding the selected memory map jumper settings. ROMs jumpered for high memory mapping are mapped to their corresponding positions in low memory.

Switch positions 8 and 9 select the automatic self-tests and I/O port loopback tests; position 9 includes the ROM test. You must install loopback connectors on the I/O ports for these tests.

Switch position 10 selects the automatic self-tests and then causes the KXT11–CA to wait for self-test commands through the TPR from the arbiter.

The *KXT11–CA Single-Board Computer User's Guide* and *KXJ11–CA Single-Board Computer User's Guide* provide further information.

On a KXJ11–CA, switch position 7 selects the automatic self-tests and I/O port loopback tests.

Note

On a KXJ11–CA, do not use boot/self-test positions 8–15 if you are using MicroPower/Pascal.

B.3 KX/KK Device Driver Communication Protocol

This section describes the protocol that the KX and KK device drivers use to communicate with one another through the 2-port RAM (TPR). It contains information to assist you in designing your own KX or KK device driver, so it can communicate with the DIGITAL–supplied KX or KK driver. The KX and KK drivers are used when the KXT11–CA or KXJ11–CA is set up for peripheral processor operation.

The protocol provides a master-slave relationship between the arbiter processor and the peripheral processor. (Do not confuse master-slave with the bus-master/bus-slave hardware concept.) The KK driver running on the peripheral processor uses the TPR to emulate a traditional Q-bus peripheral device. The KX driver running on the arbiter communicates with this device. The protocol implements a request-reply dialog between the arbiter on the Q-bus and the peripheral processor to ensure correct and complete transfer of data between them.

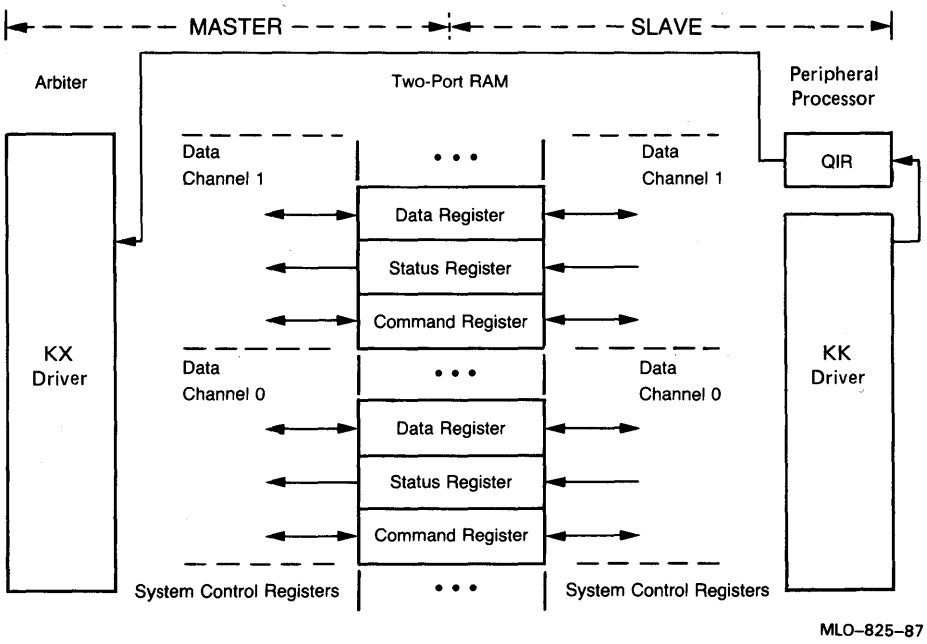
The arbiter is the master in all transactions with the peripheral processor, which is the slave (see Figure B–6). The peripheral processor must receive a command from the arbiter before passing any data to the arbiter or before receiving any data from the arbiter.

B.3.1 Communication Mechanisms

The basic TPR hardware communication mechanisms are:

- Command register for each data channel—used by the master to pass commands to the slave
- Status register for each data channel—used by the slave to pass error and operational status to the master
- Data registers—4 bytes for data channel 0 and 12 bytes for data channel 1—used for passing data between the master and the slave
- QIR register in the slave—used by the slave to interrupt the master

Figure B–6: KX/KK Device Driver Communication Linkage



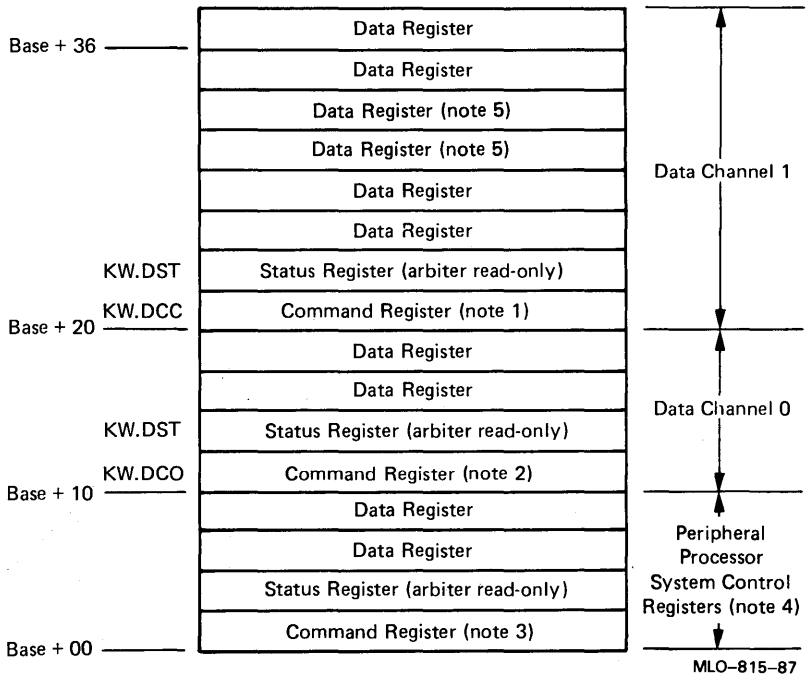
The interface between the arbiter and the peripheral processor consists of layers of software. The lowest layer contains MACRO definitions for the bits in the command and status registers of the TPR. The next layer consists of the KX and KK drivers that move data between a peripheral processor process and a process in the arbiter.

The protocol provides the arbiter with commands that cause the following:

- Device initialization of the peripheral processor
- Arbiter read request/peripheral processor write reply sequence
- Arbiter write request/peripheral processor read reply sequence
- Interrupts from the peripheral processor to the arbiter to be enabled or disabled

The following sections describe the special meanings the protocol assigns to the TPR registers in the context of KX/KK driver operations. Figure B–7 shows the TPR's general layout.

Figure B–7: TPR Register Layout



Note 1 Writing to this register from the arbiter causes a level 5 interrupt through vector 124 on the peripheral processor.

Note 2 Writing to this register from the arbiter causes a level 5 interrupt through vector 120 on the peripheral processor.

Note 3 Writing to this register from the arbiter causes the peripheral processor to restart the native firmware.

Note 4 The system control registers are not part of the protocol. They are used by the KUI utility program, diagnostic programs, and possibly user-written programs.

Note 5 Generally, the TPR at base+30 is a command register for a third data channel and the TPR at base+32 is a status register for that channel. For the KX/KK protocol, the second and third data channels are combined to form a larger data channel. To make this work, the TPR interrupts for this channel must be disabled.

B.3.2 KX/KK Protocol Definition

In the protocol, a data channel's command register (KW.DCO in Figure B–7) controls ownership of the contents of the data channel's data register. If the command register is zero, the KX driver owns the data channel's registers. If the command register is nonzero, the KK driver owns them. A data channel's status register (KW.DST in Figure B–7) is owned by the KK driver. The KX driver can only read the status registers.

If the KX driver communicates with the KK driver with interrupts disabled, the KX driver must poll the command register (KW.DCO), using it as the ownership flag for the data channel. To poll the KK driver, the KX driver must:

1. Poll the command register until it becomes zero. The zero condition means the KK driver is idle and any previous command has been completed.
2. Fill in the data registers as necessary (for example, for a write). Then issue a command by writing it into the command register. This procedure causes the KK driver to execute the command. Once the command register has been written, the KX driver cannot alter the contents of any of the data channel's registers. Consequently, the KX driver must write the data being transferred by the command into the data registers before the command is issued.
3. Poll the command register until it becomes zero. At that time, the status register data reflects the status of the previous command. The KX driver should check the status register and can then issue further commands (as in step 2).

Note

In the MicroPower/Pascal implementation, the KX driver always tells the KK driver to use the mode with interrupts enabled.

If the KX driver communicates with the KK driver with interrupts enabled, the KK device driver uses the Q-bus interrupt register (QIR) to signal the KX driver that an operation has been completed. The KK driver interrupts the KX driver after a command has been completed, the proper status and/or data bits have been set, and the command register has been cleared.

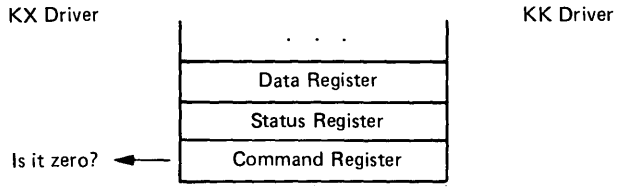
By using the interrupt-on-data-available (KC.IDA) and interrupt-on- data-requested (KC.IDR) bits of the Enable Interrupts command, the KX driver can instruct the KK driver to interrupt the KX driver when the KS.DA and/or KS.DR status register bits change from 0 to 1. This mechanism allows the KK driver to interrupt the KX driver when a user write or read request comes in on the KK side.

With interrupts enabled, the KX driver should still check the command register to make sure it has a value of 0 before filling in the data and issuing a command. However, the KX driver does not need to poll the command register after that because the KX driver interrupts when it completes executing the command.

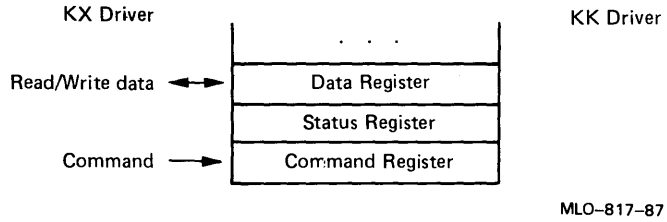
B.3.2.1 KX and KK Driver Transactions

The transactions between the KX driver and the KK driver are:

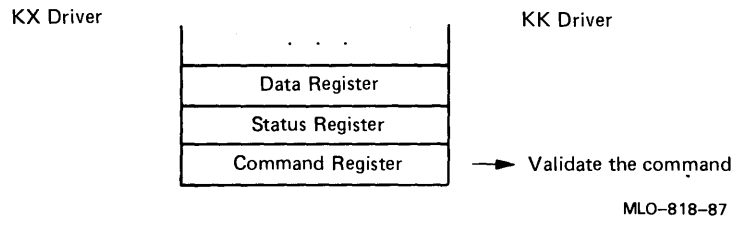
1. The KX driver tests the command register. If it is zero, the driver proceeds to the next step; if it is nonzero, the driver repeats this step.



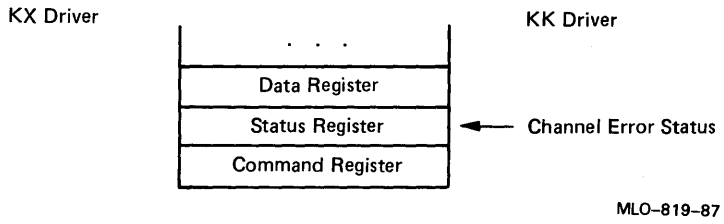
2. Since the command register is zero, the KX driver owns the data channel and can write data to or read data from the data registers, as implied by the pending command. The KX driver issues the command by writing it to the command register. The act of writing the command to the command register, thereby making it nonzero, switches ownership of the data channel to the KK driver.



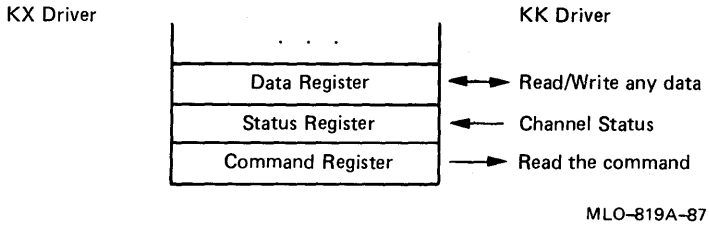
3. The KK driver is interrupted by the KX driver writing to the command register. The KK driver now owns the data channel. It reads and validates the command. During this time, if interrupts are enabled, the KX driver can wait for another user request.



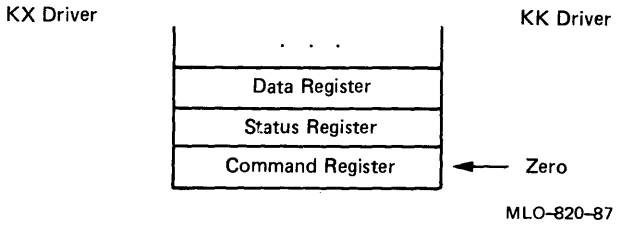
If the KK driver detects an error in the command, it reports this error in the status register and proceeds with step 4.



If the KK driver detects no error, it performs the command, moves any data required by the command into or out of the data registers, and writes the status of the channel into the status register.

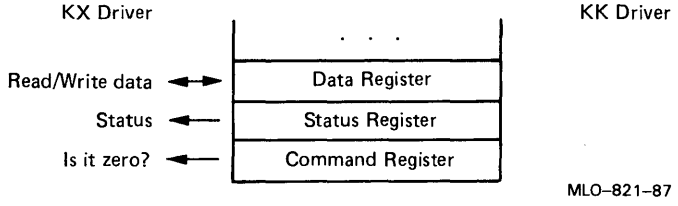


4. The KK driver completes the transaction by zeroing the command register, thus transferring ownership of the data channel back to the KX driver. If interrupts are enabled, the KK driver interrupts the KX driver by queueing an interrupt to the QIR. The vector written to the QIR is the vector that was passed in the enable interrupts command used to initialize the interface.



5. Finally, the KX driver regains ownership of the data channel in one of the following ways:
   - Polling the command register and testing for zero
   - Waiting for an interrupt and testing the command register on being interrupted to find it is zero

   Once it gains ownership, the KX driver checks the status register for error and status information and reads from or writes to the data registers, as implied by the pending command. From this point the cycle repeats.



B.3.2.2 Message Communication Between the KX and KK Drivers

The KX driver controls the passing of messages. Data is passed across the TPR interface on a channel only when the KX driver issues a read data or write data command. Before any user read or write requests are made to the KX driver, the KK driver must be running (that is the peripheral processor has started) or an error occurs. For each user request on the KX side, there must be a corresponding user request on the KK side. If there is a user read request on the KX side, there must be a corresponding user write request on the KK side and vice versa.

In the transactions between the KX and KK drivers over the TPR (Section B.3.2.1), the number of bytes in a data transfer is limited by the number of data registers in the channel (4 bytes for data channel 0 and 12 bytes for data channel 1). However, at the user application-program level of communication, the protocol provides for longer messages by using an end-of-message (EOM) indicator. Thus, a user read or write request to the KX or KK driver causes multiple transactions over the TPR if the message is larger than the size of the data channel.

When the KX driver receives a user write request from the arbiter program, it performs as many TPR write data operations as necessary to send the entire message. The byte count in the command for all operations except the last is the size of the data channel. For each KX write data operation, the KK driver completes the transaction by moving the data from the TPR channel's data registers to the user's buffer. On the last write operation, the KX driver sets the EOM indicator in the data channel's command register. The byte count for the last write is the number of bytes remaining in the message. The EOM indicator informs the KK driver that all data has been sent for this user request. Therefore, the arbiter program's user write request and the peripheral processor program's user read request are both complete.

When the KX driver receives a user read request from the arbiter program, it performs as many read data operations as necessary to receive the entire message. For each KX read data operation, the KK driver completes the transaction by moving data from the user's buffer to the TPR data registers. The byte count in the command register for all operations except the last is the size of the data channel. When the last data from the user's buffer is placed in the data registers, the KK driver sets the EOM indicator in the data channel's status register. The byte

count for the last read is the number of bytes left in the message. The EOM indicator informs the KX driver that all data has been sent for this user request. Therefore, the arbiter program's user read request and the peripheral processor program's user write request are both complete.

Normally, you should make sure that the number of bytes in the user read request on one side equals the number of bytes in the corresponding user write request on the other side. However, if the user write request from either side is for fewer bytes than the user read request on the other side, the number of bytes received is less than the number of bytes specified in the user's read request. If the user's write request from either the arbiter program or the peripheral processor program writes more data than the corresponding user read request specifies on the other side, a data overrun error occurs. For user read requests on the KX side, when the system detects a data overrun, the KX driver issues a reset command to terminate the user's write request on the KK side.

B.3.2.3 Synchronizing KX and KK Device Driver Operations

To synchronize TPR operations, the KX and KK drivers use three interrupts:

- On command completion
- When data is available (indicating that the KK driver has a pending user write request)
- When data is requested (indicating that the KX driver has a pending user read request)

These interrupts are issued by the KK driver to the KX driver. With the last two interrupts enabled, the KX driver does not issue a write data command for a user write request, unless the data-requested bit is set in the TPR status register; nor does it issue a read data command for a user read request unless the data-available bit is set in the TPR status register. In either case, it waits for an interrupt to come in from the KK driver, indicating that the appropriate user request is pending on the KK side.

The interrupts for data available and data requested come in asynchronously to the KX driver, which means that the KX driver cannot tell when the interrupt comes in. A potential for race conditions occurs if a data-available interrupt comes in while the KX driver is filling in the TPR registers to start a write, or if a data-requested interrupt comes in while the KX driver is filling in the TPR registers to start a read. To avoid the possibility of the driver's filling in the registers part way, having an interrupt come in, and then the interrupt service routine's writing over the partially filled TPR area, use the following method:

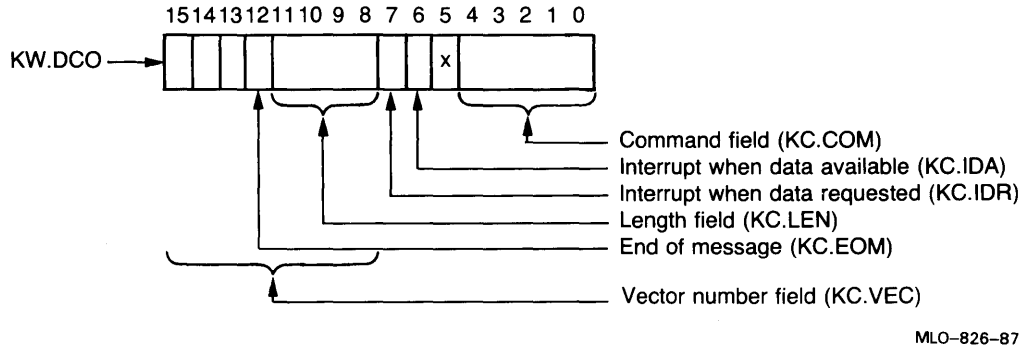
1. Issue all read data and write data commands from the interrupt service routine. This method precludes the possibility of the interrupt service routine's writing over a partially filled-in TPR channel.
2. When a user read or write request comes in on the KX side, start the transaction by issuing from driver level either an enable interrupts command (if interrupts are not yet enabled) or a Get Status command (if interrupts are enabled). In either case, the KK driver executes the command and interrupts back to the KX driver when the command finishes. The interrupt service routine, which is entered when the interrupt comes in, can then issue the first read data or write data command to continue the read or write request.

Note

The Get Status command is preferred in this instance because a No-op is a 0 command, which can cause protocol problems: 0 in the command register normally indicates that the KX driver has control.

From this point on, the rest of the message is sent, in pieces dictated by the size of the TPR data area, by the KX interrupt service routine when an interrupt comes in from the KK driver. This interrupt typically is for completion of the previous command. The KX driver interrupt service routine can check the data-requested bit in the TPR status word and a local flag to find out whether to continue a write. The local flag indicates whether a write is in progress. Similarly, the interrupt service routine can check the data-available bit and a local flag, to find out whether a read is in progress, to know whether to continue a read. The following sections define the command and status registers. The definitions are defined in the source of the KX driver (KX.MAC).

B.3.3 Command Register Definition



Bit 5 is always set to 0. Bits KC.IDA, KC.IDR, and KC.EOM and fields KC.LEN and KC.VEC have meaning for specific commands only.

B.3.3.1 Command Field (KC.COM)

The command field, KC.COM, contains one of the following commands, issued to the KK driver from the KX driver. You can use the codes (which are in decimal) in your program as word indexes in a table.

No-op Command KC$NOP (Code 0)

A null operation. The KK driver places the data channel's status in the status register (KW.DST) and clears the command register (KW.DCO). If interrupts are enabled (KC$EI command), a Q-bus interrupt occurs.

Reset KK Driver to KX Driver Command KC$RSM (Code 2)

Resets ownership of the TPR to the KX driver. The KK driver reports the channel's status in the status register (KW.DST) and clears the command register (KW.DCO). If interrupts are enabled (KC$EI command), a Q-bus interrupt occurs.

This command is used to complete a user read request on the KX side and the corresponding user write request on the KK side that terminates because of an error, for example, buffer overflow. This procedure notifies the KK driver that it should complete its write request and that there are no more read data commands for this read/write request.

Enable Interrupt Command KC$EI (Code 4)

Enables interrupt mode in the KK driver. The address of the vector to use must be in the KC.VEC field (bits 8 to 15) of the command register and must be the vector address divided by four.

After this the KK driver interrupts the KX driver when it finishes executing a command. The KK driver sets the interrupt-enabled bit (KS.IEN) in the status register when the driver executes this command.

If the interrupt-when-data-available bit (KC.IDA) is also set when this command is issued, the KK driver also interrupts the KX driver whenever new data is available (when a new user write request is processed on the KK side). The KK driver uses a local flag to keep track of whether this mode is active.

If the interrupt-when-data-requested bit (KC.IDR) is also set when this command is issued, the KK driver also interrupts the KX driver whenever new data is requested (when a new user read request is processed on the KK side). The KK driver uses a local flag to keep track of whether this mode is active.

Disable Interrupt Command KC$DI (Code 6)

Disables interrupt mode in the KK driver. The KK driver places the channel's status in the status register, including clearing the interrupt-enabled bit (KS.IEN), and clears the command register. Completion of the command never interrupts the KX driver. After this, the KK driver does not interrupt the KX driver unless an enable interrupt command is issued.

Get Status Command KC$GS (Code 8)

Instructs the KK driver to place its internal status in the data registers. This status information is currently undefined and reserved by DIGITAL. The command is effectively the same as the No-op command. The KK driver places the channel status in the status register and clears the command register. If interrupts are enabled (KC$EI command), a Q-bus interrupt occurs.

Read Status Command KC$SS (Code 10)

Instructs the KK driver to read the new internal status from the data registers. This status information is currently undefined and reserved by DIGITAL. The command is effectively the same as the No-op command. The KK driver places the channel's status in the status register and clears the command register. If interrupts are enabled (KC$EI command), a Q-bus interrupt occurs.

Read Data Command KC$RD (Code 12)

Causes the KK driver to place bytes of data into the channel's data registers. The maximum number of bytes to transfer is specified by the data length field KC.LEN (see Section B.3.3.4). The KK driver must be ready to send the data (as indicated by the data-available KS.DA bit in the status register) or it will return a no-data-available (KE$NDA) error in the status register. This indicates a protocol error if the interrupt-when-data-available mode is being used. In that mode, the KX driver does not issue this command when a user read request is made on the KX side if the data-available (KS.DA) bit is not set. Instead, the KX driver issues the read data command only after the KK driver receives a user write request and interrupts with the data-available (KS.DA) bit set.

The KK driver:

- Moves the data into the data registers
- Sets the number of bytes being transferred in the actual length field (KS.ALN) of the status register
- Sets the end-of-message bit (KS.EOM) in the status register if this is the last transfer in the user message
- Sets any other status
- Clears the command register and interrupts the arbiter if interrupts are enabled

If the user buffer being filled with data on the KX side overflows, the KX driver reports a buffer overflow error and issues a reset KK-driver-to-KX-driver command (KC$RSM) thereafter.

Write Data Command KC$WD (Code 14)

Causes the KK driver to accept bytes of data from the channel's data registers. The maximum number of bytes to transfer is specified by the data length field KC.LEN (see Section B.3.3.4). The KK driver must be ready to accept the data (as indicated by the data-requested KS.DR bit in the status register) or it returns the no-data-requested (KE$NDR) error in the status register. This procedure indicates a protocol error if the interrupt-when-data-requested mode is being used. In that mode, the KX driver does not issue this command when a user write request is made on the KX side if the data-requested (KS.DR) bit is not set. Instead, the KX driver issues the write data command only after the KK driver receives a user read request and interrupts with the data-requested (KS.DR) bit set. If the user buffer being filled with data on the KK side overflows, excess data is discarded and the KK driver returns a buffer overflow (KE$OVR) error status.

The KK driver:

- Examines the data length field (KC.LEN) for the number of bytes being transferred
- Removes the number of bytes specified by KC.LEN from the data registers
- Tests for the EOM bit (KC.EOM) to find if this is the last transfer in the message
- Places its status in the status register
- Clears the command register and interrupts the arbiter if interrupts are enabled

B.3.3.2 Interrupt-When-Data-Available Bit (KC.IDA)

When set to 1, the interrupt-when-data-available bit, KC.IDA, indicates that the KK driver should interrupt the KX driver when the data-available bit (KS.DA) of the status register changes from 0 to 1. The KK driver sets the KS.DA bit when a new user write request is processed on the KK side. The bit is cleared when a write request has been completed unless another write request is pending. The KC.IDA bit is meaningful only when used with the KC$EI command.

B.3.3.3 Interrupt-When-Data-Requested Bit (KC.IDR)

When set to 1, the interrupt-when-data-requested bit, KC.IDR, indicates that the KK driver should interrupt the KX driver when the data-requested bit (KS.DR) in the status register changes from 0 to 1. The KK driver sets the KS.DR bit when a new user read request is processed on the KK side. The bit is cleared when a read request has been completed unless another read request is pending. The KC.IDR bit is meaningful only when used with the KC$EI command.

B.3.3.4 Data Length Field (KC.LEN)

The data length field, KC.LEN, indicates the maximum number of bytes to be transferred by a read-data (KC$RD) or write-data (KC$WD) command. This field is meaningful only when used with the KC$RD and KC$WD commands. The maximum length is the number of bytes which can fit in the data registers for the channel (4 for channel 0 and 12 for channel 1) or the number of bytes remaining in the message, if the number of bytes remaining is less than the number of bytes which can fit in the data registers for the channel.

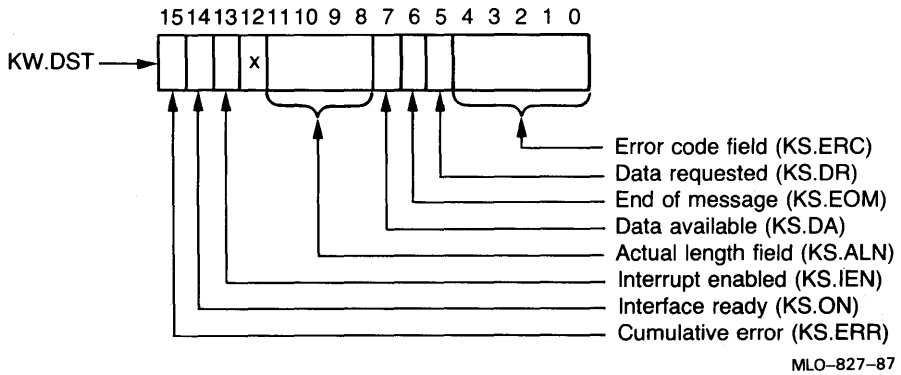
B.3.3.5 End-of-Message Bit (KC.EOM)

When set to 1, the end-of-message bit, KC.EOM, indicates that the last byte in the current write ends the user message. This bit is meaningful only when used with the KC$WD command. A corresponding bit in the status word is set by the KK driver for KX read commands (KC$RD) to indicate that the last byte in the current KK write ends the user message.

B.3.3.6 Vector Number Field (KC.VEC)

The vector number field, KC.VEC, specifies the vector number (vector address divided by four) of the interrupt vector being activated by the enable interrupt command (KC$EI). Subsequently, this vector is written to the QIR register by the KK driver when it interrupts the KX driver. This field is meaningful only when used with the KC$EI command. Each channel on each peripheral processor on the system must have a unique vector.

B.3.4 Status Register Definition



Bit 12 is set to 0.

B.3.4.1 Error Code Field (KS.ERC)

The error code field, KS.ERC, contains the following status or one of the following errors after a requested command operation. The codes are designed to let your program index them from a table. When a code other than success (KE$OK) is set, the cumulative error bit (KS.ERR) is also set. The KX driver can check the KS.ERR bit for any error and then only needs to check the particular error codes if that bit is set.

Operation Successful Status KE$OK (Code 0)

The operation previously requested completed without errors.

No Data Available Error KE$NDA (Code 2)

The read data command (KC$RD) was rejected, because no data was available. This rejection indicates the occurrence of a protocol error if the interrupt-when-data-available mode is being used.

No Data Requested Error KE$NDR (Code 4)

The write data command (KC$WD) was rejected, because no data was requested by the KK driver. This rejection indicates the occurrence of a protocol error if the interrupt-when-data-requested mode is being used.

Illegal Command Error KE$ILC (Code 6)

The command specified in the command field (KC.COM) of the command register is invalid.

Illegal Length Error KE$ILL (Code 8)

The number of bytes specified in the data length field (KC.LEN) of the command register is invalid. The length is zero or it exceeds the number of bytes in the data registers for the channel.

Illegal Vector Error KE$ILV (Code 10)

The vector number (vector address divided by four) specified in the vector number field (KC.VEC) of the command register is invalid.

KK Driver Buffer Overflow Error KE$OVR (Code 12)

The user buffer being filled by a write data (KC$WD) command on the KK side overflowed, and excess data was discarded.

B.3.4.2 Data-Requested Bit (KS.DR)

When set to 1, the data-requested bit, KS.DR, indicates the KK driver is requesting data. Thus, the KK driver is ready to execute a write data (KC$WD) command issued by the KX driver. The KK driver sets this bit when a new user read request is processed. The KK driver clears the bit when the user read request has been completed, unless another read request is pending.

B.3.4.3 End-of-Message Bit (KS.EOM)

When set to 1, the end-of-message bit, KS.EOM, indicates that the last byte in the current transfer ends the message. This bit is meaningful only for ending user read requests on the KX side to indicate that the corresponding user write request on the KK side has been completed.

B.3.4.4 Data-Available Bit (KS.DA)

When set to 1, the data-available bit, KS.DA, indicates data is available to be read from the KK driver. Thus, the KK driver is ready to execute a read data (KC$RD) command issued by the KX driver. The KK driver sets this bit when a new user write request is processed. The KK driver clears the bit when the user write request has been completed, unless another write request is pending.

B.3.4.5 Actual Length Field (KS.ALN)

The actual length field, KS.ALN, is set to the number of bytes actually transferred in response to a KX driver read data (KC$RD) or write data (KC$WD) command.

B.3.4.6 Interrupt-Enabled Bit (KS.IEN)

When set to 1, the interrupt-enabled bit, KS.IEN, incates an enable interrupt (KC$EI) command completed successfully, and the KK driver interrupts the arbiter for the specified interrupt condition(s).

B.3.4.7 Interface-Ready Bit (KS.ON)

When set to 1, the interface-ready bit, KS.ON, indicates to the KX driver that the KK driver is ready to perform the protocol.

B.3.4.8 Cumulative-Error Bit (KS.ERR)

When set to 1, the cumulative-error bit, KS.ERR, indicates an error condition exists. The error code is in the error code (KS.ERC) field. When this bit is set to 1, the KS.ALN field is not meaningful, and you should ignore its contents.

B.3.5 Interface Initialization

At system start-up, the TPR is locked from write access by the KX driver side, and the status and command registers are in a cleared state. The KX driver waits for the KK driver to initialize itself and indicate its readiness by setting the interface-ready bit (KS.ON) in the status register to 1. The KK driver cannot clear the KS.ON bit until it has permanently ceased TPR communication.

B.4 KXT11-CA and KXJ11-CA CSR and Vector Assignments

This section lists the interrupt vector assignments for all KXT11-CA devices and their associated CSRs. See Section B.5 for the interrupt vector and CSR assignments for the TPR.

Vector	CSR Address	Device	Comments
60	177560–177562	SLU1 console DLART receiver	
64	177564–177566	SLU1 console DLART transmitter	
70	175700–175716	SLU2 hardware	Do not specify this vector as an argument to the MicroPower/Pascal DEVICES macro. The kernel routes its interrupts from this vector through vectors 140 to 174.
	175720–175736	8254 timer 0 and timer 1	Timer 0 and timer 1 on the 8254 device provide timing for SLU2.
100	177520	Line frequency clock	Interrupts through this vector are enabled in the MicroPower/Pascal kernel if CLOCK=ON in the KXT11C macro. The MicroPower/Pascal clock driver enables the interrupt with or without specifying CLOCK=ON. CSR 177520 is KXTCSRA. Bit 6 enables/disables the line frequency clock as in the usual clock CSR at 177546. However, the other bits are allocated to serve other functions.
104	175720–175736	8254 timer 2	MicroPower/Pascal does not support this timer. You must write your own driver for it. Specify vector 104 in the MicroPower/Pascal DEVICES macro. Timer 2 is enabled by bit 7 in KXTCSRA (address 177520)
120	175010–175016	TPR data channel 0	This vector is used when the arbiter writes to TPR word 4 (command register for data channel 0).
124	175020–175036	TPR data channel 1	This vector is used when the arbiter writes to TPR word 8 (command register for data channel 1).
	177532	QIR	Q-bus interrupt register. The MicroPower/Pascal KK device driver writes the arbiter's vector address to use for TPR operations.
130	—	Q-BUS interrupt answer-back	This vector is used when the arbiter acknowledges the interrupt requested by the MicroPower/Pascal KK device driver over the QIR.

Vector	CSR Address	Device	Comments
134	175030–175036	TPR data channel 2	This vector is used when the arbiter writes to word 12 of the TPR (enabled by a bit in KXTCSRD). Words 12 to 15 of the TPR form data channel 2, which is not supported by the MicroPower/Pascal KK/KX device driver. Instead MicroPower/Pascal uses these locations as the last four words of data channel 1. Do not specify this vector in the MicroPower/Pascal DEVICES macro. Module KSLU2 in MicroPower/Pascal's kernel fans out the interrupts from SLU2, the multiprotocol chip, through the vector at 70 to the vectors 140 to 174.
140	175700–175736	SLU2 channel A character received	
144	175700–175736	SLU2 channel A character sent	
150	175700–175736	SLU2 channel A error	
154	175700–175736	SLU2 channel A modem control	
160	175700–175736	SLU2 channel B character received	
164	175700–175736	SLU2 channel B character sent	
170	175700–175736	SLU2 channel B error	
174	175700–175736	SLU2 channel B modem control	
200	177000–177140	Parallel I/O port A	This vector is set up by the KXT11-CA/KXJ11-CA native firmware and used by parallel I/O port A.
204	177000–177140	Parallel I/O port B	This vector is set up by the KXT11-CA/KXJ11-CA native firmware and used by parallel I/O port B.
210	177000–177140	Parallel I/O timers	This vector is set up by the KXT11-CA/KXJ11-CA native firmware and used by parallel I/O port counter-timers.

Vector	CSR Address	Device	Comments
220	—	Arbiter RESET	This vector is used when the arbiter's BRESET signal is asserted. BRESET is asserted when the arbiter executes a RESET instruction or its RESTART switch is toggled. Many device interrupts are enabled and disabled by setting bits in the CSRs.
224	174400–174536	DTC channel 0	The native firmware sets up this vector.
230	174400–174536	DTC channel 1	The native firmware sets up this vector.
	175000–175006	TPR system control	The first four words of the TPR used for KXT11–CA or KXJ11–CA native firmware/arbiter communication. When the arbiter writes to word 0 (TPR command register), the KXT11–CA restarts at 173004.

B.5 System ID Switch Positions, Two-Port RAM CSR and Vector Assignments

This section shows the CSR and interrupt vector assignments for the TPRs that are selected by the KXT11–CA or KXJ11–CA system ID switch. These registers and vectors appear in the I/O page and vector area of arbiter memory. The table also shows the associated KX device driver logical unit IDs.

The KX device driver passes data between the arbiter CPU and up to 14 peripheral processors running on the Q-bus. The driver communicates with the KK device driver in the KXT11–CA or KXJ11–CA through the command and status registers in the data channel areas of the TPR. (See Section B.3 for more information.)

ID Switch Position	MicroPower KX Driver ID	TPR Base Jumper In=KXT Out=KXJ	Address Jumper Out=KXT In=KXJ	Default Vectors MicroPower
0	Stand-alone mode			
1	Stand-alone mode			
2	A	17762100	17760100	500,504
3	B	17762140	17760140	510,514
4	C	17762200	17760200	520,524
5	D	17762240	17760240	530,534
6	E	17762300	17760300	540,544
7	F	17762340	17760340	550,554

ID Switch Position	MicroPower KX Driver ID	TPR Base Jumper In=KXT Out=KXJ	Address Jumper Out=KXT In=KXJ	Default Vectors MicroPower
8	G	17777400	17775400	560,564
9	H	17777440	17775440	570,574
10	I	17777500	17775500	600,604
11	J	17777540	17775540	610,614
12	K	17777600	17775600	620,624
13	L	17777640	17775640	630,634
14	M	17777700	17775700	640,644
15	N	17777740	17775740	650,654

B.6 Sample MicroPower/Pascal Configuration File

The following is a copy of the MicroPower/Pascal configuration file that you must edit according to your hardware and software requirements in preparation for building a KXT11–CA application.

```
      .enabl   LC
;+
; Configuration file for a KXT11--CA target
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1984, 1986 BY DIGITAL EQUIPMENT CORPORATION.  ALL RIGHTS RESERVED.
;+
;
; Module name: CFDKTC.MAC
;
; System: Micropower/Pascal
;
; Modified:
;
;      RLP     23-Apr-86        Changed DMA vectors to match KXJ.
;
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;      CONFIGURATION  [version-name]
;      SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;      PROCESSOR mmu={YES|NO},[fpu={FP11|FIS|FPA}],
;                type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C|CMR21},
;                j11map={YES|NO},[vector=nnnn],
;                clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;         (Vector default is 1000 octal for an L11x or J11 target type,
;          400 octal for the other target types.)
;              Note: Standard clock CSRs, if present, are:
;                    For an LSI11/23-PLUS or J-11 = 177546
;                    For a KXT11--CA or KXJ11--CA   = 177520
;                    Default is no clock csr. Do not specify clkcsr unless
;                    there is a clock csr.
```

```
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;           Parameters can be:
;           ALL    - All primitives (default for p1,...,p6)
;           BCSEM  - Binary and counting semaphore primitives
;           COMPLX - Complex primitives
;           EXCMGT - Exception handling primitives
;           INTMGT - Interrupt handling primitives
;           LOGNAM - Logical name primitives
;           DRAM   - Region allocation, sharing, and mapping primitives
;           PRMGT  - Process management primitives
;           QSEMN  - Nonprivileged queue-semaphore primitives
;           QSEMP  - Privileged queue-semaphore primitives
;           RBUF   - Ring buffer primitives
;           STRMGT - Structure management primitives
;           TIMER  - Clock service primitives
;           V1     - All V1 primitives
;           xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;           Parameters can be:
;           ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;           TR4 - Trap to 4 (bus timeout)
;           T10 - Trap to 10 (reserved instruction)
;           BPT - Breakpoint instruction trap
;           EMT - EMT instruction trap
;           TRP - TRAP instruction trap
;           MPT - Memory parity error
;           FIS - FIS exception trap
;           FPP - FPP exception trap
;           MMU - Memory management fault
;           BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL name, string
;   ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
        .enabl  GBL

        .mcall  CONFIGURATION

        .sbttl  System Configuration File For KXT11--CA Target

CONFIGURATION
```

```
SYSTEM          debug=YES, optimize=YES          ; ADDRCHECK defaults to DEBUG
                                                 ; value

PROCESSOR       mmu=NO, type=KXT11C, vector=400, clock=60HZ, clkcsr=177520

MEMORY          base=0, size=511., type=RAM
; Assumes 32KB of volatile native RAM (map 0): Note that the highest 64 bytes
; of the native RAM (1 memory block) must not be described here, since it is
; used by the native firmware and therefore is not allocatable.

KXT11C          bhalt=YES, reset=IGNORE, map=0  ;Factory map configuration

RESOURCES       packets=10., structures=2048.   ;Small pools for packets
                                                ;and kernel structures
PRIMITIVES      ALL

TRAPS           ALL                     ;Implies T4, T10, BPT, EMT, and TRP

DEVICES         60,64,100               ;Console serial line (SLU1)
                                        ;and clock vectors

; DEVICES       104                     ;Spare timer vector, if used

; DEVICES       224,230                 ;DMA vectors

DEVICES         120,124,130             ;Two-Port RAM arbiter write interrupts

DEVICES         140,144,150,154         ;SLU2 pseudo-vectors - channel A
DEVICES         160,164,170,174         ;SLU2 pseudo-vectors - channel B

; DEVICES       200,204,210             ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXT11C macro
; DEVICES       220                     ;Simulated QBUS reset-interrupt vector

ENDCFG

.end
```

B.7 Sample Configuration Files for the KXJ11-CA

```
        .enabl   LC
;+
; Configuration File For Unmapped KXJ11--CA Target
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFDKJU.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;       CONFIGURATION  [version-name]
;       SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;       PROCESSOR mmu={YES|NO},[fpu={FP11|FIS|FPA}],
;                 type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C|CMR21},
;                 j11map={YES|NO},[vector=nnnn],
;                 clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;          (Vector default is 1000 octal for an L11x or J11 target type,
;           400 octal for the other target types.)
;              Note: Standard clock CSRs, if present, are:
;                    For an LSI11/23-PLUS or J-11 = 177546
;                    For a KXT11--CA or KXJ11--CA   = 177520
;                    Default is no clock csr. Do not specify clkcsr unless
;                    there is a clock csr.
;       MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;              volatile={YES|NO},res={YES|NO}[,name=string]
;       DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;       RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;       PRIMITIVES p1,p2,p3,p4,p5,p6
;                Parameters can be:
;                ALL    - All primitives (default for p1,...,p6)
;                BCSEM  - Binary and counting semaphore primitives
;                COMPLX - Complex primitives
;                EXCMGT - Exception handling primitives
;                INTMGT - Interrupt handling primitives
;                LOGNAM - Logical name primitives
;                DRAM   - Region allocation, sharing, and mapping primitives
;                PRMGT  - Process management primitives
;                QSEMN  - Nonprivileged queue-semaphore primitives
```

```
;               QSEMP  - Privileged queue-semaphore primitives
;               RBUF   - Ring buffer primitives
;               STRMGT - Structure management primitives
;               TIMER  - Clock service primitives
;               V1     - All V1 primitives
;               xxxx   - where xxxx is a specific primitive name (no $)
;
;       Required if processor type is FALC or FALCPLUS --
;       FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;       Required if processor type is KXT11C --
;       KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;       Required if processor type is KXJ11C --
;       KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;       TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;               Parameters can be:
;               ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;               TR4 - Trap to 4 (bus timeout)
;               T10 - Trap to 10 (reserved instruction)
;               BPT - Breakpoint instruction trap
;               EMT - EMT instruction trap
;               TRP - TRAP instruction trap
;               MPT - Memory parity error
;               FIS - FIS exception trap
;               FPP - FPP exception trap
;               MMU - Memory management fault
;               BRK - FALCON (SBC--11/21) BREAK level-7 trap
;       LOGICAL   name, string
;       ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
        .enabl   GBL
        .mcall   CONFIGURATION
        .sbttl   System Configuration File For Unmapped KXJ11--CA Target

CONFIGURATION

SYSTEM      debug=YES, optimize=YES      ; ADDRCHECK defaults to DEBUG
                                         ; value
PROCESSOR   mmu=NO, type=KXJ11C, vector=400, clock=60HZ, clkcsr=177520

;Leave 128. bytes just below 160000 for the firmware stack
MEMORY      base=0, size=<28.*32.-2>, type=RAM
; Uses a total of 56KB of volatile native RAM - 128(10) bytes

KXJ11C      bhalt=NO, reset=IGNORE

RESOURCES   packets=10., structures=2048.   ;Small pools for packets
                                            ;and kernel structures
PRIMITIVES   ALL

TRAPS        ALL              ;Implies T4, T10, BPT, EMT, and TRP
```

```
DEVICES      60,64,100          ;Console serial line (SLU1)
                                ;and clock vectors
; DEVICES    104                ;Spare timer vector, if used

DEVICES      120,124,130        ;Two-Port RAM arbiter write interrupts

DEVICES      140,144,150,154    ;SLU2 pseudo-vectors - channel A
DEVICES      160,164,170,174    ;SLU2 pseudo-vectors - channel B

; DEVICES    200,204,210        ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES    220                ;Simulated QBUS reset-interrupt vector

; DEVICES    224,230            ;DMA vectors

ENDCFG

.end
       .enabl   LC
;+
; Configuration File for Mapped KXJ11--CA Target without J11 mapping support;
; 32 KW of memory. Also includes the PIO chip vectors.
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFDKJM.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;      CONFIGURATION  [version-name]
;      SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;      PROCESSOR mmu={YES|NO},[fpu={FP11|FIS}],
;                type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C},
;                [vector=nnnn],
;                clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;         (Vector default is 1000 octal for an L11x or J11 target type.)
;              Note: Standard clock CSRs, if present, are:
;                    For an LSI11/23-PLUS or J-11 = 177546
;                    For a KXT11--CA or KXJ11--CA   = 177520
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
```

```
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
;      QSEMP  - Privileged queue-semaphore primitives
;      RBUF   - Ring buffer primitives
;      STRMGT - Structure management primitives
;      TIMER  - Clock service primitives
;      V1     - All V1 primitives
;      xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;      Parameters can be:
;      ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;      TR4 - Trap to 4 (bus timeout)
;      T10 - Trap to 10 (reserved instruction)
;      BPT - Breakpoint instruction trap
;      EMT - EMT instruction trap
;      TRP - TRAP instruction trap
;      MPT - Memory parity error
;      FIS - FIS exception trap
;      FPP - FPP exception trap
;      MMU - Memory management fault
;      BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL   name, string
;   ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
   .enabl   GBL

   .mcall   CONFIGURATION
```

```
   .sbttl   System Configuration File for Mapped KXJ11--CA Target
CONFIGURATION
SYSTEM      debug=YES, optimize=NO      ; ADDRCHECK defaults to DEBUG
                                        ; value
PROCESSOR   mmu=YES, type=KXJ11C, j11map=no, vector=400, clock=60HZ, clkcsr=177520
;Leave a hole of 128. bytes for the firmware stack just below 160000
; Uses 64KB of volatile native RAM total
MEMORY      base=0, size=<<28.*32.>-2>, type=RAM
MEMORY      base=<28.*32.>, size=<4.*32.>, type=RAM

KXJ11C      bhalt=YES, reset=IGNORE

DEVICES     60,64,100           ;Console serial line (SLU1)
                                ;and clock vectors

; DEVICES   104                 ;Spare timer vector, if used

DEVICES     120,124,130         ;Two-Port RAM arbiter write interrupts

DEVICES     140,144,150,154     ;SLU2 pseudo-vectors - channel A
DEVICES     160,164,170,174     ;SLU2 pseudo-vectors - channel B

DEVICES     200,204,210         ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES   220                 ;Simulated QBUS reset-interrupt vector

;DEVICES    224,230             ;DMA vectors
ENDCFG
.end

   .enabl   LC
;+
; Configuration File for Mapped KXJ11--CA Target with J11 Mapping Support.
; Also includes all 512 KB of memory.
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFDKJJ.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;   CONFIGURATION  [version-name]
```

```
;   SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;   PROCESSOR mmu={YES|NO},[fpu={FP11|FIS|FPA}],
;        type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C|CMR21},
;        j11map={YES|NO},[vector=nnnn],
;        clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;      (Vector default is 1000 octal for an L11x or J11 target type,
;       400 octal for the other target types.)
;      Note: Standard clock CSRs, if present, are:
;            For an LSI11/23-PLUS or J-11 = 177546
;            For a KXT11--CA or KXJ11--CA   = 177520
;            Default is no clock csr. Do not specify clkcsr unless
;            there is a clock csr.
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
;      QSEMP  - Privileged queue-semaphore primitives
;      RBUF   - Ring buffer primitives
;      STRMGT - Structure management primitives
;      TIMER  - Clock service primitives
;      V1     - All V1 primitives
;      xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;      Parameters can be:
;      ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;      TR4 - Trap to 4 (bus timeout)
;      T10 - Trap to 10 (reserved instruction)
;      BPT - Breakpoint instruction trap
;      EMT - EMT instruction trap
;      TRP - TRAP instruction trap
;      MPT - Memory parity error
;      FIS - FIS exception trap
;      FPP - FPP exception trap
;      MMU - Memory management fault
;      BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL   name, string
;   ENDCFG
;
```

```
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
   .enabl   GBL

   .mcall   CONFIGURATION

   .sbttl   System Configuration File for Mapped KXJ11--CA Target

CONFIGURATION

SYSTEM      debug=YES, optimize=NO      ; ADDRCHECK defaults to DEBUG
                                        ; value
;Defaults to J11MAP=yes for type=KXJ11C or J11
;PROCESSOR   mmu=YES, type=KXJ11C, vector=400, clock=60HZ, clkcsr=177520
; Uses a total 512KB of volatile native RAM
;Must leave a hole for the native firmware stack area (128. bytes starting at
;157600)
;MEMORY      base=0, size=<28.*32.-2>, type=RAM
;Note: base could also be specified as 1600(8), size as 16200(8).
MEMORY      base=<28.*32.>, size=<228.*32.>, type=RAM
KXJ11C      bhalt=NO, reset=IGNORE
DEVICES     60,64,100           ;Console serial line (SLU1)
                                ;and clock vectors
; DEVICES   104                 ;Spare timer vector, if used

DEVICES     120,124,130         ;Two-Port RAM arbiter write interrupts

DEVICES     140,144,150,154     ;SLU2 pseudo-vectors - channel A
DEVICES     160,164,170,174     ;SLU2 pseudo-vectors - channel B

;DEVICES    200,204,210         ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES   220                 ;Simulated QBUS reset-interrupt vector
;DEVICES    224,230             ;DMA vectors

ENDCFG

.end
```

```
;File CFR8KJM.MAC - for 27128 chips.
   .enabl   LC
;+
; Configuration File for Mapped KXJ11--CA ROM/RAM Target without J11 mapping
; support. With 27128 ROM chips.
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFR8KJM.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;   CONFIGURATION  [version-name]
;   SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;   PROCESSOR mmu={YES|NO},[fpu={FP11|FIS}],
;        type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C},
;        [vector=nnnn],
;        clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;      (Vector default is 1000 octal for an L11x or J11 target type.)
;      Note: Standard clock CSRs, if present, are:
;            For an LSI11/23-PLUS or J-11 = 177546
;            For a KXT11--CA or KXJ11--CA   = 177520
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
;      QSEMP  - Privileged queue-semaphore primitives
;      RBUF   - Ring buffer primitives
;      STRMGT - Structure management primitives
```

```
;       TIMER  - Clock service primitives
;       V1     - All V1 primitives
;       xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;      Parameters can be:
;      ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;      TR4 - Trap to 4 (bus timeout)
;      T10 - Trap to 10 (reserved instruction)
;      BPT - Breakpoint instruction trap
;      EMT - EMT instruction trap
;      TRP - TRAP instruction trap
;      MPT - Memory parity error
;      FIS - FIS exception trap
;      FPP - FPP exception trap
;      MMU - Memory management fault
;      BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL   name, string
;   ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
   .enabl   GBL
   .mcall   CONFIGURATION
   .sbttl   System Configuration File for Mapped KXJ11--CA Target

CONFIGURATION

SYSTEM      debug=NO, optimize=YES      ; ADDRCHECK defaults to DEBUG
                                        ; value
PROCESSOR   mmu=YES, type=KXJ11C, j11map=no, vector=400, clock=60HZ, clkcsr=177520
;User ROM at 2000000(8) to 2037777(8) is also visible at 0(8) - 37777(8)
;MEMORY      base=0, size=400, type=ROM

;This shows the same thing using n*32. notation instead
;MEMORY      base=0, size=<8.*32.>, type=ROM

;Leave a hole of 128. bytes for the firmware stack just below 160000
;MEMORY      base=1000, size=1576, type=RAM
;This shows the same thing using n*32. notation instead
;MEMORY      base=<16.*32.>, size=<<12.*32.>-2>, type=RAM

;Now go from 160000(8) to the top of RAM memory - 1777777(8)
;MEMORY      base=1600, size=16200, type=RAM
;This shows the same thing using n*32. notation instead
;MEMORY      base=<28.*32.>, size=<228.*32.>, type=RAM
```

```
;ROM at 2000000(8) - 2037777(8) already configured at 0 - 37777(8). Above
;that, at 2040000(8) - 2077777(8) is the firmware. Then the same user code
;which is at 2000000(8) is visible again at 2100000(8) - 2137777(8), followed
;by the firmware again at 2140000(8) - 2177777(8).

KXJ11C      bhalt=NO, reset=IGNORE

RESOURCES   packets=10., structures=2048.   ;Small pools for packets
                                            ;and kernel structures
PRIMITIVES  ALL
TRAPS       ALL                 ;Implies T4, T10, BPT, EMT, and TRP
DEVICES     60,64,100           ;Console serial line (SLU1)
                                ;and clock vectors
; DEVICES   104                 ;Spare timer vector, if used

DEVICES     120,124,130         ;Two-Port RAM arbiter write interrupts

DEVICES     140,144,150,154     ;SLU2 pseudo-vectors - channel A
DEVICES     160,164,170,174     ;SLU2 pseudo-vectors - channel B

;DEVICES    200,204,210         ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES   220                 ;Simulated QBUS reset-interrupt vector

;DEVICES    224,230             ;DMA vectors

ENDCFG

.end


;File CFRKJM.MAC
   .enabl   LC
;+
; Configuration File For Mapped KXJ11--CA ROM/RAM Target without J11 mapping
; support. With 27256 ROM chips.
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFRKJM.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
```

```
;   CONFIGURATION  [version-name]
;   SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;   PROCESSOR mmu={YES|NO},[fpu={FP11|FIS}],
;        type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C},
;        [vector=nnnn],
;        clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;      (Vector default is 1000 octal for an L11x or J11 target type.)
;      Note: Standard clock CSRs, if present, are:
;            For an LSI11/23-PLUS or J-11 = 177546
;            For a KXT11--CA or KXJ11--CA   = 177520
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
;      QSEMP  - Privileged queue-semaphore primitives
;      RBUF   - Ring buffer primitives
;      STRMGT - Structure management primitives
;      TIMER  - Clock service primitives
;      V1     - All V1 primitives
;      xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;      Parameters can be:
;      ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;      TR4 - Trap to 4 (bus timeout)
;      T10 - Trap to 10 (reserved instruction)
;      BPT - Breakpoint instruction trap
;      EMT - EMT instruction trap
;      TRP - TRAP instruction trap
;      MPT - Memory parity error
;      FIS - FIS exception trap
;      FPP - FPP exception trap
;      MMU - Memory management fault
;      BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL   name, string
;   ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
```

```
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-
   .enabl   GBL
   .mcall   CONFIGURATION
   .sbttl   System Configuration File For Mapped KXJ11--CA Target

CONFIGURATION

SYSTEM      debug=NO, optimize=YES      ; ADDRCHECK defaults to DEBUG
                  ; value
PROCESSOR   mmu=YES, type=KXJ11C, j11map=no, vector=400, clock=60HZ, clkcsr=177520

;ROM at 2000000(8) to 2077777(8) is also visible at 0(8) - 77777(8)
MEMORY      base=0, size=1000, type=ROM
;This shows the same thing using n*32. notation instead
;MEMORY      base=0, size=<16.*32.>, type=ROM

;Leave a hole of 128. bytes for the firmware stack just below 160000
MEMORY      base=1000, size=1576, type=RAM
;This shows the same thing using n*32. notation instead
;MEMORY      base=<16.*32.>, size=<<12.*32.>-2>, type=RAM

;Now go from 160000(8) to the top of RAM memory - 1777777(8)
MEMORY      base=1600, size=16200, type=RAM
;This shows the same thing using n*32. notation instead
;MEMORY      base=<28.*32.>, size=<228.*32.>, type=RAM

;ROM at 2000000(8) - 2077777(8) already configured at 0 - 77777(8).
;Get the rest of the ROM - starting at 2100000(8). Firmware resides from
;2140000(8) to 2177777(8).
MEMORY      base=21000, size=400, type=ROM

;This shows the same thing using n*32. notation instead
;MEMORY      base=<272.*32.>, size=<8.*32.>, type=ROM

KXJ11C      bhalt=NO, reset=IGNORE
RESOURCES   packets=10., structures=2048.   ;Small pools for packets
                                            ;and kernel structures
PRIMITIVES   ALL

TRAPS        ALL                    ;Implies T4, T10, BPT, EMT, and TRP

DEVICES      60,64,100              ;Console serial line (SLU1)
                                    ;and clock vectors
; DEVICES   104                     ;Spare timer vector, if used

DEVICES     120,124,130             ;Two-Port RAM arbiter write interrupts

DEVICES     140,144,150,154         ;SLU2 pseudo-vectors - channel A
DEVICES     160,164,170,174         ;SLU2 pseudo-vectors - channel B

;DEVICES    200,204,210             ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES   220                     ;Simulated QBUS reset-interrupt vector

;DEVICES    224,230                 ;DMA vectors

ENDCFG

.end
```

B.8 Shared Memory on a KXJ

KXJ shared memory is memory on the KXJ that, when enabled (as shared memory), becomes visible on the Q-bus. The KXJ shared memory is then directly accessible from both the KXJ and the arbiter. In addition, this memory is then accessible by means of the DMA chip on other peripheral processors (KXT11-CAs or KXJ11-CAs). KXJ shared memory facilitates the sharing of data between the arbiter and the KXJ. The starting Q-bus address for the KXJ shared memory can be set up at any address so that the entire range of addresses is available on the Q-bus side.

Normally, the KXJ shared memory is started right above the configured memory on the Q-bus. The size of this memory can be any multiple of 4KW up to 256KW minus the size of the KXJ application. The KXJ shared memory is at the top of memory on the KXJ side.

Example

You enable KXJ shared memory starting at 1000000(8) on the Q-bus with a size of 24KW (140000(8) bytes). The KXJ shared memory is visible at locations 1000000 to 1137777 on the arbiter side. That memory is visible on the KXJ side at a start address of 2000000 - 140000 = 1640000; so it is visible at addresses 1640000 to 1777777.

Use the procedure KXJ_ENABLE_SHARED to enable KXJ shared memory, and the procedure KXJ_DISABLE_SHARED to disable KXJ shared memory. These procedures are called from the KXJ side only. The following files on the MicroPower/Pascal distribution kit are required for using these procedures:

Name	Description
KXJSHR.PAS	KXJ11-CA shared memory procedures module
MISC.PAS	Miscellaneous procedures include-file

To use a source module, you must compile it and then merge it with the program at user-process build time. The associated include-file must be included in the program at compile time.

To use KXJ shared memory:

1. Set up a MicroPower/Pascal shared region on the arbiter side and another one on the KXJ side that corresponds to the KXJ shared memory area you intend to use.

   You need to know the start address and size on the Q-bus side. Once you know the size, it fixes the start address on the KXJ side at 2000000(8) minus the size. If you don't know the needed size, pick a size large enough for all possible needs, but less than the size of the KXJ application.

   Use the appropriate MEMORY macro in the kernel configuration file on each side—specify start, size, res=YES, and name. (The names on the two sides can be different.) At system start-up time, the kernel on each side allocates the region and creates a shared region of the specified name.

2. Enable the KXJ shared memory by using the KXJ_ENABLE_SHARED procedure. If the Q-bus start address and size are known at build time, the KXJ application uses those parameters to enable shared memory. Otherwise, the arbiter determines the parameters and communicates them to the arbiter by means of the KX/KK communication interface.

3. Once the shared memory is enabled, each side can access its MicroPower/Pascal shared region by using the name specified in the MEMORY macro. You then use the MAP_WINDOW procedure (or the MAPW$ primitive from MACRO-11) to map to the shared region (which is also the KXJ shared memory).

   On the arbiter side, if the processor supports caching, the accessing process must disable cache in each PAR that maps to KXJ shared memory by specifying caching:=disable in the MAP_WINDOW primitive call. This is necessary because accesses by the KXJ to the KXJ shared memory do not invalidate the arbiter cache.

4. To synchronize access to KXJ shared memory, pass messages between the arbiter and the KXJ by using the KX/KK communication interface.

5. When you have finished using KXJ shared memory, you can disable it by using the KXJ_DISABLE_SHARED procedure.

   If you have more than one KXJ and plan to use KXJ shared memory on them, you can either set up one MicroPower shared region in the arbiter to include all the KXJ shared memory for all KXJs, or you can set up a separate MicroPower/Pascal shared region to correspond to each KXJ shared memory.

B.8.1 KXJ_ENABLE_SHARED

Enables KXJ shared memory at the specified Q-bus starting address for the specified size. The KXJ shared memory is at the top of memory on the KXJ side. The range of Q-bus addresses must be available.

Syntax

```
KXJ_ENABLE_SHARED (start,size,status)
```

Parameter	Type	Description
start	LONG_INTEGER	Q-bus physical starting address in bytes. Must be a multiple of 4KW (20000(8)).
size	LONG_INTEGER	Size of shared region in bytes. Must be a multiple of 4KW (20000(8)). Must be less than 2000000(8) minus size of KXJ application.
status	var status	Optional parameter for return of status information.

Error Returns

ES$ILV Either the start or size is an illegal value—not a multiple of 4KW; or the size is greater than or equal to 2000000(8).

No error is reported if the KXJ shared memory overlaps the KXJ application. However, this may cause unpredictable results since the arbiter application may corrupt the KXJ application.

B.8.2 KXJ_DISABLE_SHARED

Disables KXJ shared memory, making it no longer visible on the Q-bus. Successive KXJ_ENABLE_SHARED calls perform implicit KXJ_DISABLE_SHARED calls before reenabling KXJ shared memory. You need not perform a KXJ_DISABLE_SHARED first.

Syntax

```
KXJ_DISABLE_SHARED (status)
```

Parameter	Type	Description
status	var status	Optional parameter for return of information.

Error Returns

None.

B.8.3 Arbiter and KXJ Configuration Files and Applications

The following files show an arbiter configuration file, an arbiter application, a KXJ configuration file, and a KXJ application. The arbiter sends the Q-bus start address 1200000(8) and size 20000(8) to the KXJ by means of the KX/KK communication interface. The KXJ application enables shared memory and tells the arbiter the status of the operation. The arbiter then initializes the KXJ shared memory area with the values 1 to 4096 and tells the KXJ it is ready for it to increment the values. The KXJ increments the values by 1 and notifies the arbiter that it has completed the operation. The arbiter checks the values and, if they are correct, tells the KXJ to increment the values again. This process continues until the KXJ has incremented all the values 100 times. Then the arbiter tells the KXJ to disable shared memory. After that, the entire operation is repeated, starting with the enabling of KXJ shared memory.

```
     -Config file for arbiter side - shared memory example
;File cfdmash2.mac - arbiter side for shared memory test. Includes kx support.
;Use a MPP static shared region for the KXJ shared memory area.
   .enabl   LC
;+
; Configuration file for a mapped LSI--11/23 target
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1984, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFDMASH2.MAC

;
; System: Micropower/Pascal
;
```

```
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;
;   CONFIGURATION  [version-name]
;   SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;   PROCESSOR mmu={YES|NO},[fpu={FP11|FIS}],
;        type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C},
;        [vector=nnnn],
;        clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;      (Vector default is 1000 octal for an L11x or J11 target type.)
;      Note: Standard clock CSRs, if present, are:
;            For an LSI11/23-PLUS or J-11 = 177546
;            For a KXT11--CA or KXJ11--CA   = 177520
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
;      QSEMP  - Privileged queue-semaphore primitives
;      RBUF   - Ring buffer primitives
;      STRMGT - Structure management primitives
;      TIMER  - Clock service primitives
;      V1     - All V1 primitives
;      xxxx   - where xxxx is a specific primitive name (no $)
;
;   Required if processor type is FALC or FALCPLUS --
;   FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;   Required if processor type is KXT11C --
;   KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;   Required if processor type is KXJ11C --
;   KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
```

```
;   TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;      Parameters can be:
;      ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;      TR4 - Trap to 4 (bus timeout)
;      T10 - Trap to 10 (reserved instruction)
;      BPT - Breakpoint instruction trap
;      EMT - EMT instruction trap
;      TRP - TRAP instruction trap
;      MPT - Memory parity error
;      FIS - FIS exception trap
;      FPP - FPP exception trap
;      MMU - Memory management fault
;      BRK - FALCON (SBC--11/21) BREAK level-7 trap
;   LOGICAL   name, string
;   ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-

   .enabl   GBL
   .mcall   CONFIGURATION
   .sbttl   System Configuration File For Mapped LSI--11/23 Target
CONFIGURATION
SYSTEM      debug=YES, optimize=NO      ; ADDRCHECK defaults to DEBUG
                                        ; value
; Optimize=NO implies the defaults for RESOURCES, PRIMITIVES, and TRAPS
macros
PROCESSOR   mmu=YES, type=L1123, clock=60HZ
MEMORY      base=0, size=<32.*32.>, type=RAM    ;Assumes 32K words of
                                                ;volatile RAM
;MPP shared region for KXJ shared memory area. 128 KW at starting at
;1200000(8). This is assumed to be the maximum size region we will ever need
;in the application.
MEMORY      base=12000, size=<128.*32.>, type=RAM, res=YES, name=ARBSHR
DEVICES     60,64,100,300,304     ;Vectors for console terminal, clock,
                                  ;and a second serial-line unit
DEVICES     500,504               ;KX driver units.
ENDCFG
.end

-User program for arbiter side - shared memory example
{file tshra2.PAS - arbiter side.}
[ SYSTEM(MICROPOWER), PRIORITY(50),
  DATA_SPACE(2100), STACK_SIZE (200)] PROGRAM tshra2;
{Have the KXJ set enable an area of shared memory, initialize the first
 4 KWs, tell the KXJ to update it, then check it. Do the update and
 check 100 times. Then have the KXJ disable the shared memory. Then
 enable the shared region again, and so on. Always with the same start
 address (1200000(8)) and size (40000(8)). The MPP shared region is
 created at build time.}
```

```
{$NOLIST}
%include 'micropower$lib:escode.pas'    { get exception codes}
%include 'micropower$lib:misc.pas'      { get shared memory routine}
%include 'micropower$lib:kxinc.pas'     { get KX routines}
%include 'micropower$lib:dram.pas'      { get dynamic ram routines}
{$LIST}

CONST
  message_length = 6;
  big_array_size = 4096;

TYPE
  big_array = array [1..big_array_size] of integer;
 {Define fields in Message which contain the Qbus starting address and the
   size, both in bytes}
  other = record
        a : integer;
        addr : long_integer;
        addr1 : long_integer;
        d : integer;
        end;

VAR
  message : array [1..message_length] of integer;
        {First word contains the command:
                1 = enable shared memory
                2 = ok to access the region and add 1 to each value
                3 = disable shared memory.

        On an enable shared memory command, the second and third words are for
        passing the Qbus starting address for the shared memory area in bytes.
        The fourth and fifth words are for passing the size of the shared
        memory area in bytes.

   The last word is for status returns:
           1 = success
           2 = enable shared memory failed
           3 = protocol error
           4 = end of test
   }
my_exc_status : exc_status;
  actual_length,rwstatus : unsigned;
  i,j,k,error : integer;
  test_array : big_array;
  q : ^big_array;
  MY_RIB : region_id_block;
  start_addr : long_integer;
  start,size_region : unsigned;
  start_region : unsigned;
  ready : CHAR;
  readyb : boolean;

{Main program}
BEGIN
```

```
      readyb := false;
      while not readyb do
        begin
            writeln ('Type R when you are ready to begin');
            readln (ready);
            if (ready="R") or (ready="r") then
                readyb := true;
        end;
      while true do
        begin
{Set start and size of shared region}
          start_addr := %o'1200000';
          message::other.addr := %o'1200000';
          start_region := ushort ((message::other.addr) div 64);
          message::other.addr1 := %o'40000';
          size_region := ushort ((message::other.addr1) div 64);
          message[1] := 1;
          rwstatus := kx_write_data (buffer := message,
                                     length := 2*message_length,
                                     ret_length := actual_length);
          IF rwstatus <> es$nor THEN
            begin
              WRITELN ('Write of shared memory command failed');
              stop
            end
          else
              begin
                rwstatus := kx_read_data (buffer := message,
                                          length := 2*message_length,
                                          ret_length := actual_length);
                if message [message_length] <> 1 then
                   begin
                     writeln ('Enable shared memory failed');
                     stop;
                   end
                else
                  begin

                     access_shared_region (rib := my_rib,
                                           name := 'ARBSHR');
                     map_window (rib := my_rib,
                                 caching := disable,   {disable caching}
                                 length := size(big_array),
                                 offset := 0,
                                 window_ptr := q);
                     for j := 1 to big_array_size do
                        begin
                           test_array[j] := j;
                           q^[j] := j;
                        end;
```

```
                    for j := 1 to 100 do
                       begin
                         for k := 1 to big_array_size do
                               test_array[k] := test_array[k]+1;
                    message[1] := 2;
                    message[message_length] := 0;
                    rwstatus := kx_write_data (buffer := message,
                                  length := 2*message_length,
                                  ret_length := actual_length);
                    rwstatus := kx_read_data (buffer := message,
                                   length := 2*message_length,
                                   ret_length := actual_length);
                    if message[message_length] <> 1 then
                       begin
                          writeln ('Error on KXJ access   ',j);
                          writeln ('Error is: ',message[message_length]);
                       end
                    else
                       begin
                         error := 0; {Clear error count}
                         for i := 1 to big_array_size do
                             if q^[i] <> test_array[i] then
                                  error := error + 1;
                         if error <> 0 then
                          begin
                            writeln ('Error on pass   ',j,'. ',error,' errors.');
                            stop;
                          end;
                       end;
                      END;
                    writeln ('All 100 accesses passed.');
                    message[1] := 3; {disable shared memory}
                    message[message_length] := 0;
                    rwstatus := kx_write_data (buffer := message,
                                        length := 2*message_length,
                                        ret_length := actual_length);
                    rwstatus := kx_read_data (buffer := message,
                                  length := 2*message_length,
                                  ret_length := actual_length);
                    if message[message_length] <> 1 then
                           begin
                             writeln ('Error disabling shared memory.');
                             stop;
                           end;
                end;
              unmap_window (length := size(big_array),
                                window_ptr := q);
        end;
    end;
end.
```

```
     -Config file for the KXJ side for shared regions
;File CFDTSHRL3.MAC - create static shared region RESSHR for KXJ shared
memory
;of size 128 KW
   .enabl   LC
;+
; Configuration File For Mapped KXJ11--CA Target without J11 mapping support
;-
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
;+
;
; Module name: CFDTSHRL3.MAC
;
; System: Micropower/Pascal
;
; Functional Description:
;
; This module describes a hardware and system software configuration in
; which an application is to run.  The file must be edited by the user to
; reflect a specific application environment and then be assembled.  The
; resulting object module is used to build the kernel.
;
; The following set of macros may be used in a configuration file. The
; CONFIGURATION macro must be the first macro evoked.  The ENDCFG macro must
; be the last.  A configuration file must contain at a minimum the
; CONFIGURATION, SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
; In the following condensed syntax descriptions, brackets ([...]) indicate
; optionality, braces ({...}) enclose alternatives, and single parameter
; values shown in optional arguments indicate defaults.
;

;   CONFIGURATION  [version-name]
;   SYSTEM optimize={YES|NO},debug={YES|NO},addrcheck={YES|NO}
;   PROCESSOR mmu={YES|NO},[fpu={FP11|FIS}],
;        type={L112|L1123|FALC|FALCPLUS|J11|KXT11C|KXJ11C},
;        [vector=nnnn],
;        clock={NONE|50HZ|60HZ|100HZ|800HZ},[clkcsr=nnnnnn]
;      (Vector default is 1000 octal for an L11x or J11 target type.)
;      Note: Standard clock CSRs, if present, are:
;            For an LSI11/23-PLUS or J-11 = 177546
;            For a KXT11--CA or KXJ11--CA   = 177520
;   MEMORY base=nnn,size=mmm,type=ROM|RAM,parity={YES|NO},[csr=nnnnnn],
;          volatile={YES|NO},res={YES|NO}[,name=string]
;   DEVICES   vectaddr1,vectaddr2,...,vectaddr6
;   RESOURCES [stack=..KIS],[packets=20.],[structures=3000.],[ramtbl=20.]
;   PRIMITIVES p1,p2,p3,p4,p5,p6
;      Parameters can be:
;      ALL    - All primitives (default for p1,...,p6)
;      BCSEM  - Binary and counting semaphore primitives
;      COMPLX - Complex primitives
;      EXCMGT - Exception handling primitives
;      INTMGT - Interrupt handling primitives
;      LOGNAM - Logical name primitives
;      DRAM   - Region allocation, sharing, and mapping primitives
;      PRMGT  - Process management primitives
;      QSEMN  - Nonprivileged queue-semaphore primitives
```

```
;       QSEMP  - Privileged queue-semaphore primitives
;       RBUF   - Ring buffer primitives
;       STRMGT - Structure management primitives
;       TIMER  - Clock service primitives
;       V1     - All V1 primitives
;       xxxx   - where xxxx is a specific primitive name (no $)
;
;    Required if processor type is FALC or FALCPLUS --
;    FALCON trap140={BHALT|NXM},break={ROMODT|SFWODT|EXCEPTION|IGNORE|HANG}
;
;    Required if processor type is KXT11C --
;    KXT11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT},map=n
;
;    Required if processor type is KXJ11C --
;    KXJ11C bhalt={YES|NO},reset={IGNORE|BOOT|RSTBOT|INTRPT}
;
;    TRAPS  t1,t2,t3,t4,t5,t6,t7,t8
;       Parameters can be:
;       ALL - TR4, T10, BPT, EMT, and TRP (standard LSI--11 set)
;       TR4 - Trap to 4 (bus timeout)
;       T10 - Trap to 10 (reserved instruction)
;       BPT - Breakpoint instruction trap
;       EMT - EMT instruction trap
;       TRP - TRAP instruction trap
;       MPT - Memory parity error
;       FIS - FIS exception trap
;       FPP - FPP exception trap
;       MMU - Memory management fault
;       BRK - FALCON (SBC--11/21) BREAK level-7 trap
;    LOGICAL   name, string
;    ENDCFG
;
; If the value of the SYSTEM macro optimize argument is YES, the RESOURCES,
; TRAPS, and PRIMITIVES macros are required.  If the optimize argument value is
; NO (default), the RESOURCES, TRAPS, and PRIMITIVES macros are defaulted and
; should not appear in the configuration file.
;
;-

    .enabl   GBL
    .mcall   CONFIGURATION
    .sbttl   System Configuration File For Mapped KXJ11--CA Target

CONFIGURATION

SYSTEM      debug=YES, optimize=YES      ; ADDRCHECK defaults to DEBUG
                                         ; value
PROCESSOR   mmu=YES, type=KXJ11C, j11map=no, vector=400, clock=60HZ,
            clkcsr=177520

; Uses 64KB of volatile native RAM total in low memory
;Leave a hole of 128. bytes for the firmware stack just below 160000
MEMORY      base=0, size=<<28.*32.>-2>, type=RAM

MEMORY      base=<28.*32.>, size=<4.*32.>, type=RAM

;Create a shared region from 1000000(8) to 1777777(8), at the top of RAM
;memory. Use this for mapping the shared memory which is enabled on the Qbus.
;It is set up as the maximum size of shared memory which might ever get enabled
;in the application (128 KW).
MEMORY      base=10000, size=<128.*32.>, type=RAM, res=YES, name=RESSHR
```

```
KXJ11C      bhalt=YES, reset=IGNORE

RESOURCES   packets=10., structures=2048.   ;Small pools for packets
                                            ;and kernel structures
PRIMITIVES   ALL

TRAPS        ALL                    ;Implies T4, T10, BPT, EMT, and TRP

DEVICES      60,64,100              ;Console serial line (SLU1)
                                    ;and clock vectors
; DEVICES    104                    ;Spare timer vector, if used

DEVICES      120,124,130            ;Two-Port RAM arbiter write interrupts

DEVICES      140,144,150,154        ;SLU2 pseudo-vectors - channel A
DEVICES      160,164,170,174        ;SLU2 pseudo-vectors - channel B

DEVICES      200,204,210            ;PIO and counter/timer vectors

; Include the following only if reset=INTRPT in KXJ11C macro
; DEVICES    220                    ;Simulated QBUS reset-interrupt vector
;DEVICES     224,230                ;DMA vectors

ENDCFG

.end


     -User program -- KXJ side
{file tshrl3s.pas - assumes MPP "shared region" is created at build
time, with the name RESSHR. We just ACCESS it here and then map to it.
Make it a big shared region - map at approproate offset to get to the
start of the Qbus shared area. Size of KXJ shared memory area may vary.
Need device access for kxj_enable_shared and kxj_disable_shared}
[ SYSTEM(MICROPOWER), PRIORITY(50),
  DATA_SPACE(2100), STACK_SIZE (200),DEV_ACCESS] PROGRAM tshrl3;
{$NOLIST}
%include 'micropower$lib:escode.pas'    { get exception codes}
%include 'micropower$lib:misc.pas'      { get shared memory routine}
%include 'micropower$lib:kkinc.pas'     { get KK routines}
%include 'micropower$lib:dram.pas'      { get dynamic ram routines}
{$LIST}

TYPE
  big_array = array [1..4096] of integer;
  other = record
        a : integer;
        addr : long_integer;
        addr1 : long_integer;
        d : integer;
        end;

CONST
  message_length = 6;
  big_array_size = 4096;
```

```
VAR
  message : array [1..message_length] of integer;
        {First word contains the command:
                1 = enable shared memory
                2 = ok to access the region and add 1 to each value
                3 = disable shared memory.
        The last word is for status returns:
                1 = success
                2 = enable shared memory failed
                3 = protocol error
                4 = end of test
        }
  my_exc_status : exc_status;
  actual_length,rwstatus : unsigned;
  i,j,bigloop : integer;
  q : ^Big_array;
  MY_RIB : region_id_block;
  start_addr,local_start_addr : long_integer;
  start : unsigned;
  shared_size : long_integer;
  shared_size_particks : integer;
  offset_into_region : unsigned;

{Main program}
BEGIN
      bigloop := 0;
      while true do
        begin
          bigloop := bigloop +1;
          rwstatus := kk_read_data (buffer := message,
                                    length := 2*message_length,
                                    ret_length := actual_length);
          IF rwstatus <> es$nor THEN
            WRITELN ('Read of shared memory info from arbiter failed')
            else
              begin
                if message[1] <> 1 then
                  begin
                    writeln ('Wrong command. Should be an enable shared memory command.');
                    message[message_LENGTH] := 3;
                  end
                else
                  begin
                     start_addr := message::other.addr;
                     shared_size := message::other.addr1;
                     shared_size_particks := ushort (shared_size div 64);
                     kxj_enable_shared(start := start_addr,
                                       size := shared_size,
                                       status := my_exc_status);
                     if my_exc_status.EXC_CODE = es$nor then
                       begin
                        writeln ('Enable shared memory   ',bigloop,' succeeded');
                        local_start_addr := %o'2000000' - shared_size;
```

```
                    message[message_length] := 1;
                  end
                 else
                  begin
                    writeln ('Enable shared memory   ',bigloop,'
                             ' failed. Status code is ',
                             oct(my_exc_status.exc_code),'octal.');
                    message[message_length] := 2;
                  end;

                 rwstatus := kk_write_data (buffer :=message,
                               length := 2*message_length,
                               ret_length := actual_length);

                 access_shared_region (rib := my_rib,
                                       name := 'RESSHR');

{May have to alter this depending on where the shared area starts
 compared to where the statically created shared region starts}
                 offset_into_region := ushort(local_start_addr div 64);
                 offset_into_region := offset_into_region -
                           my_rib.region_address;
                 map_window (rib := my_rib,
                             length := size(big_array) ,
                             offset := offset_into_region,
                             window_ptr := q);
                 for j := 1 to 100 do
                   begin
          rwstatus := kk_read_data (buffer := message,
                        length := 2*message_length,
                        ret_length := actual_length);
          if message[1] <> 2 then
             begin
                writeln ('Protocol error.');
                message [message_length] := 3;
             end
          else
             begin
               for i := 1 to big_array_size  do
                  q^[i] := q^[i] + 1;
               message[message_length] := 1;
             end;
          rwstatus := kk_write_data (buffer := message,
                        length := 2*message_length,
                        ret_length := actual_length);
          writeln ('Access   ',j,' to shared region completed.');
        end;
          rwstatus := kk_read_data (buffer := message,
                        length := 2*message_length,
                        ret_length := actual_length);
          if message[1] <> 3 then
             begin
               writeln ('Protocol error, should be disabling.');
               message[message_length] := 3;
             end
          else
             kxj_disable_shared (status := my_EXC_STATUS);
          if my_exc_status.EXC_CODE = es$nor then
            begin
              writeln ('Disable shared memory   ',
```

```
                    bigloop,' succeeded');
                message [message_length] := 1;
              end
            else
               BEGIN
                  writeln ('Disable shared memory   ',
                     bigloop,' failed. Status code is ',
                     oct(my_exc_status.exc_code),'octal.');
                  message[message_length] := 4;
               end;
            rwstatus := kk_write_data (buffer := message,
                          length := 2*message_length,
                          ret_length := actual_length);
  end;
unmap_window (length := size(big_array),
                    window_ptr := q);
      end;
   end;
end.
```

B.9 Calculating Checksums for PROMS

This section tells you how to use the VMS DECprom program to calculate checksums for PROM devices (programmable read-only memories) on the KXT11–CA. The checksums calculated by this method can be verified by the ROM checksum test performed by the KXT11–CA self-tests in the native firmware. The *KXT11–CA Single-Board Computer User's Guide* describes the algorithm that the ROM checksum test uses to calculate checksums.

DECprom calculates only checksums for PDP–11 processors based on a 16-bit system word. The ROM checksum test in the KXT11–CA native firmware expects that each PROM device will contain its own (byte) checksum. Therefore, you must calculate a separate checksum for each PROM device, then load the appropriate checksum value into the last location of each device.

The following procedure assumes you know how to use DECprom. See the *VAX/VMS DECprom User's Guide* for detailed reference information.

1. Using an initialization file that is appropriate for your PROM devices, run DECprom.
2. Set the system word width to 16 bits.
3. Load the PROM devices from the input file type of your choice (.MIM, .LDA, .SAV) but leave the last location of each PROM empty.
4. Exit DECprom.
5. Perform steps 1 and 2 (above), but set the system word width to eight bits. This allows DECprom to calculate a byte-wide checksum.
6. LIST the contents of each PROM in a binary (.SAV) file. This will produce a file for the low-byte device and a file for the high-byte device.
7. Calculate a separate checksum for each file (CALC_CHECKSUM command).
8. Store the appropriate checksum in the last location of each device (STORE_CHECKSUM command).

For the KXJ11-CA, the standard DECprom checksum algorithm is used, but with one checksum for the native firmware and another checksum for the user application, if any. See the *KXJ11-CA Single-Board Computer User's Guide* for details on how to burn ROM chips and include checksums.

B.10 Load Application onto KXT11-CA/KXJ11-CA Procedure

KXT_LOAD and KXJ_LOAD are MicroPower/Pascal procedures that run on the arbiter and are used to load and start a MicroPower/Pascal application on a KXT11-CA or KXJ11-CA respectively. The procedure reads in a .MIM file one block at a time and commands a KXT11-CA or KXJ11-CA to DMA each block into its local memory. When all blocks have been loaded into the KXT's or KXJ's memory, QBUS ODT commands are given to the KXT or KXJ thereby causing it to start the application (jump to the address contained in octal location 24).

B.10.1 .MIM File

The file that the KXT_LOAD or KXJ_LOAD procedure loads is the KXT or KXJ memory image file (.MIM) that is output by the MIB utility. To be loaded onto a KXT11-CA with KXT_LOAD, the memory image must be in an unmapped format. For a KXJ11-CA, the memory image can be either mapped or unmapped. Neither the KXT nor the KXJ memory image being loaded should contain the Debug Service Module (DSM).

The .MIM file may or may not have the TU58 boot program installed in it. The load utility will skip over and ignore the boot if it exists. Thus, a memory image loadable in two different ways can be built. The memory image can be copied to a TU58 tape and booted via the KXT or KXJ console port or it can reside on one of the arbiter's file-structured devices and be booted via the KXT_LOAD or KXJ_LOAD procedure.

B.10.2 User's Interface

You gain access to the KXT_LOAD or KXJ_LOAD procedure by including the definition file MISC.PAS in your program source. You must compile the module KXTLO.PAS (which contains the source code for both KXT_LOAD and KXJ_LOAD) and merge it with your program at user-process build time.

A process that calls the KXT_LOAD or KXJ_LOAD procedure must observe the following guidelines:

- The process must have access to the I/O page.
- The KXT_LOAD or KXJ_LOAD procedure can load only one KXT or KXJ at a time. If multiple KXT/KXJ loads are required, the calling process must either serialize them or have multiple processes calling the KXT_LOAD or KXJ_LOAD procedure.

The syntax for calling the KXT_LOAD procedure is given below. The syntax for calling the KXJ_LOAD procedure is the same as that for calling KXT_LOAD except for the obvious need to change occurrences of KXT to KXJ:

```
KXT_LOAD (KXT_ADDR,MIMF,STATUS);
```

Parameter	Type	Description
KXT_ADDR	UNSIGNED	Virtual address of KXT11-CA
VAR MIMF	[READONLY]PACKED ARRAY [L..U : INTEGER] OF CHAR	File name
VAR STATUS	EXC_STATUS	System error code optional argument

B.10.3 Program Example

The following program uses the KXT_LOAD procedure to load a sample KXT11-CA application into KXT11-CA ID 2 with the board configured for the low address range. (KXT11-CA ID numbers are listed in Section B.5.) The KXJ11-CA application is similar, with "KXJ" replacing "KXT."

```
[SYSTEM(MICROPOWER), STACK_SIZE(1000), PRIORITY(10), DATA_SPACE(3000),
DEV_ACCESS ]
     PROGRAM LOADK ;
{ Define the Load Procedure }
%include 'MISC.PAS'
{ Define Address for KXT11--CA ID 2 }
CONST

    KXT2_ADDR = %O'160100' ;

{ Define Error Return Status }
VAR
    LOAD_RESULTS : EXC_STATUS ;
{ Main Program }
BEGIN
    KXT_LOAD (KXT_ADDR := KXT2_ADDR,
      MIMF := 'DYAO:EXKXT.MIM',
      STATUS := LOAD_RESULTS);
END.
```

Appendix C
XL Serial Line Driver

This chapter describes the use of the MicroPower/Pascal XL serial line driver, which supports operations on terminals and other devices attached to serial line interfaces.

Note

In Version 2 of MicroPower/Pascal, the XL driver has been superseded by the TT driver, described in Chapter 3. However, the XL driver is supplied on the distribution kit in three versions (PDP-11 mapped, PDP-11 unmapped, and KXT11-CA) for existing applications that require it. DIGITAL recommends that applications use the TT driver.

Since the XL driver has been superseded by the TT driver, support for new hardware has not been added. Therefore, the XL driver is not supported on the KXJ11-CA.

C.1 PDP-11 XL Driver

The XL device driver supports I/O operations on devices connected through a DLV11 type of serial line interface unit. The DLV11, DLV11-E, DLV11-F and DLV11-J; the MXV11-A and MXV11-B serial lines; and the SBC-11/21 serial lines are supported by the XL driver.

The XL driver performs input and output operations in either block mode or ring mode. In block-mode input, a specified number of bytes are transferred directly from the serial line unit to the requester's buffer space, per read request. In block-mode output, a specified number of bytes are transferred from the requester's data buffer to the serial line unit, per write request. Block-mode input is provided for input devices that, presumably, transmit data in blocks, or bursts of characters, of predetermined length, with some time lapse between block transmissions.

In ring-mode input, the driver continuously transmits bytes from the serial line unit to an input ring buffer specified by the requester. Once initiated by a Connect Receive Ring Buffer request for a given unit, the input operation continues until the ring buffer is disconnected.

In ring-mode output, the driver continuously transmits bytes from an output ring buffer, as they become available, to the specified serial line unit. The requester or some other process must first put characters into the ring buffer. Following this, the process issues a Connect Transmit Ring Buffer request for a given unit, which starts the output transfer. The output operation continues until the ring buffer is disconnected. Ring mode is intended for unbounded or interactive input, such as from a terminal keyboard or an instrument that continuously monitors some fluctuating quantity—for example, a voltage or temperature.

For output requests, the driver provides optional XOFF/XON control character processing. If requested, the driver inhibits the output side of a given serial line unit when it detects an XOFF character on the input side and reenables the output when a subsequent XON character is received.

For interface units that support modem control (DLV11–E), the driver allows you to enable data-set interrupts and to receive data-set status information, analogous to RCSR contents, when such interrupts occur. The driver also allows you to get (inspect) data-set status information and to set certain XCSR and RCSR bits; this permits you to control baud rate, if programmable, to exercise modem control, and to transmit break signals.

The XL driver normally consists of two processes—a request-handling main process and a line-control subprocess. These two processes control all the serial line interface units that may be configured on the target system. In ring buffer output mode, however, a helper process is created for each line so connected, to facilitate low-overhead output operations.

The driver's request queue semaphore name is $XLA. The desired serial line is specified in the function request by a configuration-determined unit number.

Each DLV11 interface unit is factory-configured with standard device register and interrupt vector addresses. When more than one serial line unit is included in a system, or if the standard address assignments are not desired, you must reconfigure the unit(s) in question for appropriate device register and interrupt vector addresses. In any case, the device register and vector addresses used—whether standard or not—must be specified in a kernel configuration file (see Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual*) and in a driver prefix file (see Section C.1.4).

C.1.1 Functions Provided

The following functions provided by the XL driver are listed by symbolic and decimal function code:

Code	Function Performed
IF$RDP (0)	Read Physical
IF$RDL (1)	Read Logical
IF$WTP (3)	Write Physical
IF$WTL (4)	Write Logical
IF$SET (6)	Set Status
IF$GET (7)	Get Status
IF$CRR (8)	Connect Receive Ring Buffer
IF$CXR (9)	Connect Transmit Ring Buffer
IF$DRR (10)	Disconnect Receive Ring Buffer
IF$DXR (11)	Disconnect Transmit Ring Buffer
IF$RSC (12)	Report Data-Set Status Change

For serial line service, there is no distinction between physical and logical function requests; either function code may be used for block mode. One device-dependent function modifier bit is significant for an output function request—either a write or a Connect Transmit Ring Buffer function—as it enables automatic XOFF/XON control, as described below.

C.1.1.1 Read Function

The IF$RDP or IF$RDL function is performed in block mode. The driver reads a specified number of characters from the serial line unit into the buffer area identified in field DP.BUF of the request message. No device-dependent function modifiers apply to a read operation request.

Read requests to a unit that has been connected to a ring buffer are not supported.

C.1.1.2 Write Function

The IF$WTP or IF$WTL function is performed in block mode; the source and amount of the data to be transferred are specified by the DP.BUF field of the request message. One device-dependent function modifier applies to write functions, as follows:

Bit	Significance
FM$XCK (11)	Enable automatic XOFF/XON processing if set; disable same if cleared

For automatic XOFF/XON processing, the driver intercepts any XON and XOFF characters that occur in the input from a given interface unit. Receipt of an XOFF character causes the driver to inhibit output from the same interface unit until a subsequent XON character is received. The output line is initially assumed to be enabled. (Multiple successive XON or XOFF characters have the same effect as one such character.) If XOFF/XON processing is not requested, the driver passes all input characters to the user.

Write requests to a unit that has been connected to a ring buffer are not supported.

C.1.1.3 Connect Receive Ring Buffer Function

The IF$CRR function connects a user-specified ring buffer to a serial line unit and initiates input to that ring buffer. Any input occurring on the line is transferred to the ring buffer identified in request field DP.SGL; field DP.BUF of the request message is ignored. Input into the ring buffer continues until the ring buffer is disconnected. No device-dependent function modifiers apply to a Connect Receive Buffer request.

Note

When a line is connected to a ring buffer, all hardware errors are ignored, including parity errors, framing errors, and overrun errors.

C.1.1.4 Disconnect Receive Ring Buffer Function

The IF$DRR function disconnects a user-specified ring buffer from a serial line unit. Any subsequent input occurring on the line is ignored. No device-dependent function modifiers apply to a Disconnect Receive Buffer request.

This request will not be honored for any line that a ring buffer was created for and connected to as specified by the driver prefix file.

C.1.1.5 Connect Transmit Ring Buffer Function

The IF$CXR function connects a user-specified ring buffer to a serial line unit and initiates output from that ring buffer. Any data put into the ring buffer identifier in request field DP.SGL is output on the specified line. Field DP.BUF of the request message is ignored. One device-dependent function modifier applies to write functions, as follows:

Bit	Significance
FM$XCK (11)	Enable automatic XOFF/XON processing if set; disable same if cleared

XOFF/XON processing is performed as described above for block-mode operations.

C.1.1.6 Disconnect Transmit Ring Buffer Function

The IF$DXR function disconnects a user-specified ring buffer from a serial line unit. No device-dependent function modifiers apply to a Disconnect Receive Buffer request.

C.1.1.7 Report Data-Set Status Change Function

The IF$RSC function allows the requester to wait for a change of data-set (modem) status to occur on a specified serial line unit. When such a change occurs, the driver returns a standard IF$GET reply message containing status information, as described in Section C.1.1.9. This function is used to wait for signals, such as ring, lost carrier, and so forth.

When this function is requested, the driver assumes that modem control (data-set interrupts) has been enabled by a previous Set Status (IF$SET) request. If this request is issued with data-set interrupts disabled, the process will wait indefinitely. (Data-set control is meaningful only if the specified serial line unit is a DLV11-E configured for full modem control.)

For this function, you must specify a queue semaphore at DP.SGL to be signaled when a status change occurs on the specified unit. The request holds for only one event. More than one request for notification may be posted for a specified unit.

C.1.1.8 Set Status Function

The IF$SET function sets the specified interface unit's RCSR and/or XCSR with control bits that allow the following actions to occur:

1. Enable/disable modem control (DLV11-E only)
2. Load modem-status bits (DLV11-E only):
   a. Data terminal ready
   b. Request to send
   c. Secondary transmit
3. Transmit a break
4. Set programmable baud rate (DLV11-E, DLV11-F, MXV11-B, and SBC-11/21 only)

The desired status settings for the receiver status register are indicated by the bit settings of word DP.RPS of the request message. The desired status settings for the transmitter status register are indicated by the bit settings of word DP.XPS of the request message. The status-control word is described in the next section.

C.1.1.9 Get Status Function

The IF$GET function returns information about the interface unit's status. The driver reads the specified unit's RCSR and XCSR and returns selective information about its settings in the reply message. Software-specified parameters are returned for both the receiver and the transmitter. (The status information is specific to the hardware and varies accordingly.)

C.1.1.10 Device-Independent Function Modifiers

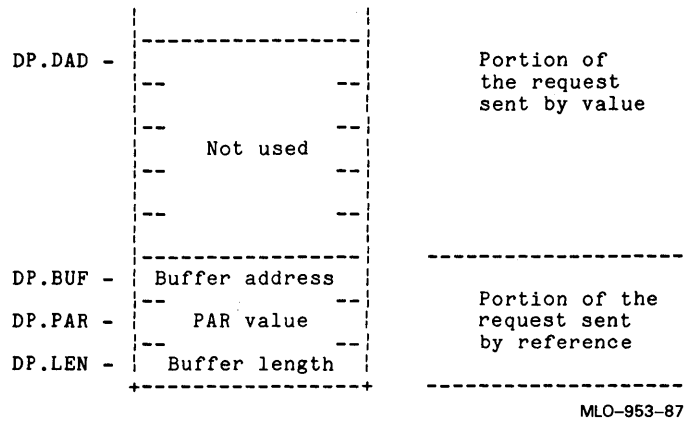
If bit FM$BSM of DP.FUN is set, the XL driver signals a binary or a counting semaphore, as described in Chapter 1. The setting of bit FM$INH of DP.FUN (inhibit soft-error retry) is not meaningful for serial line service and is ignored by the XL driver.

C.1.2 Function-Dependent Request Formats

The function-independent portion of a driver request message is described in Chapter 1. The function-dependent portion of a XL driver request (following field DP.SEM) is described below for each type of function.

C.1.2.1 Block-Mode Read or Write Functions

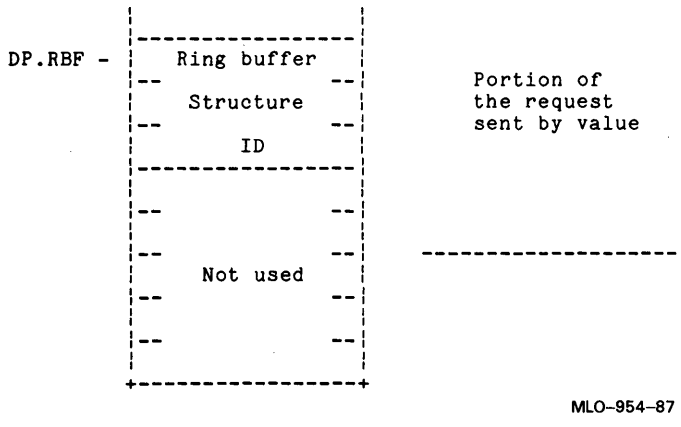
The function-dependent portion of a block-mode read request (function code IF$RDP or IF$RDL) or of a write request (function code IF$WTP or IF$WTL) is shown below:



The unit number in the function-independent portion of the request selects the desired line-interface unit; unit numbering starts at 0. The buffer address specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer.

C.1.2.2 Connect Receive or Transmit Ring Buffer Functions

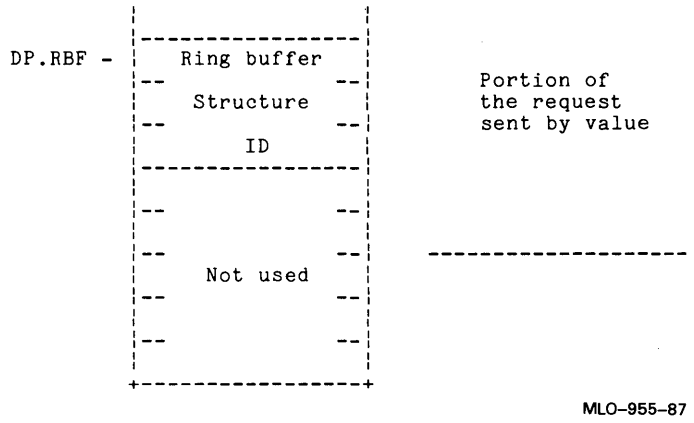
The function-dependent portion of a Ring Buffer Connect request for either input or output (function code IF$CRR or IF$CXR) is shown below:



The unit number in the function-independent portion of the request selects the desired line-interface unit; unit numbering start at 0. Field DP.RBF specifies, by structure ID, the destination ring buffer for an input operation or the source buffer for an output operation. On input, individual characters are put into the ring buffer as they are received; on output, individual characters are transmitted from the ring buffer as they become available by action of the user process. In either case, the length of the transfer is unlimited.

C.1.2.3 Disconnect Receive or Transmit Ring Buffer Functions

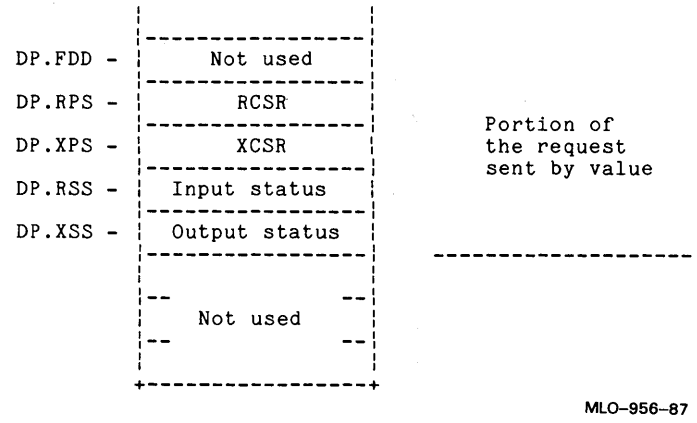
The function-dependent portion of a Ring Buffer Disconnect request, for either input or output (function code IF$DRR or IF$DXR), is shown below. Its format is identical to that for the connect request:



The unit number in the function-independent portion of the request specifies the line-interface unit from which the ring buffer is to be disconnected. Field DP.RBF specifies, by structure ID, the ring buffer that was previously connected to the unit in question. All input to or output from the ring buffer ceases when the request is acted on by the driver.

C.1.2.4 Set Status Function

The function-dependent portion of a Set Status request (function code IF$SET) is shown below:

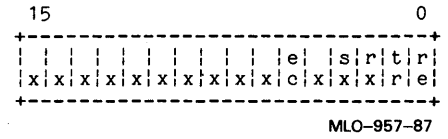


The request packet fields shown above have the following significance:

Field	Significance
DP.RPS	Status control bits to be set in the receiver CSR; these bit settings are hardware-dependent
DP.XPS	Status control bits to be set in the transmitter CSR; these bit settings are hardware-dependent
DP.RSS	Receiver software status bit settings
DP.XSS	Transmitter software status bit settings

The bit settings of DP.RPS are used to set status control bits in the RCSR. DP.XPS is used to set status control bits in the XCSR of the specified interface unit.

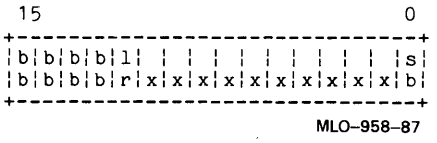
The format of the receiver status-setting word is as follows:



Proceeding from right to left in the format above:

- The re bit (0), if set, advances the paper tape reader in DIGITAL-modified TTY units (LT33–C, LT35–A,C) and clears the RCVR DONE bit in the RCSR (bit 7). The function of this bit is hardware-dependent.
- The tr bit (1), if set, indicates data terminal ready (DLV11–E only).
- The rx bit (2), if set, indicates request to send (DLV11–E only).
- The sx bit (3), if set, indicates secondary Xmit (DLV11–E only).
- The ec bit (5), if set, enables modem control (DLV11–E only).

The format of the transmitter status-setting word for the DLV11, DLV11–E, DLV11–F, DLV11–J, and MXV11–A serial line interfaces is as follows:

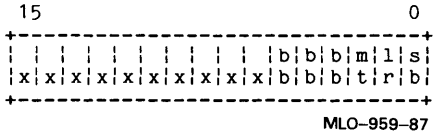


MLO–958–87

Proceeding from right to left in the format above:

- The sb bit (0), if set, requests a BREAK to be transmitted on the output line.
- The lr bit (11), if set, allows a new baud rate to be loaded.
- The bb bits (12 to 15) indicate the desired baud rate for the unit, if the lr bit is also set (valid only if the baud rate is programmable on the indicated unit).

The format of the transmitter status-setting word for the MXV11–B and SBC–11/21 serial line interfaces is as follows:

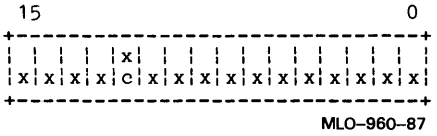


MLO–959–87

Proceeding from right to left in the format above:

- The sb bit (0), if set, requests a BREAK to be transmitted on the output line.
- The lr bit (1), if set, allows a new baud rate to be loaded.
- The mt bit (2), if set, facilitates a maintenance self-test. When this bit is set, the transmitter serial output is connected to the receiver serial input while the external serial input is disconnected.
- The bb bits (3 to 5) indicate the desired baud rate for the unit, if the lr bit is also set (valid only if the baud rate is programmable on the indicated unit).

The format of the software status-setting words is as follows:



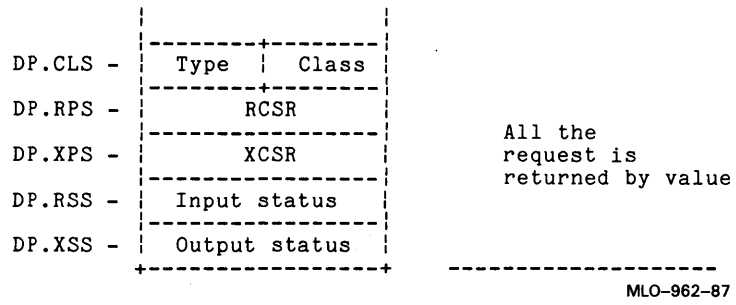
MLO–960–87

Proceeding from right to left in the format above:

- The xc bit (11), if set, requests XON/XOFF processing for the input side. No meaning is assigned to this bit for the output side.

C.1.2.5 Get Status Function

The function-dependent portion of a reply to a Get Status request (function code IF$GET) is shown below:



In the information above:

- Class is DC$TER for serial line interfaces.
- Type indicates the specific type of DLV11 interface, as follows:

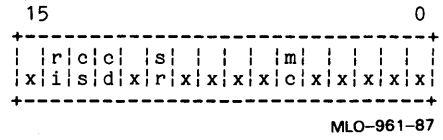
  TT$DL Minimum serial line capability (DLV11, DLV11-J, MXV11-A)

  TT$DLE DLV11-E

  TT$DLF DLV11-F

  TT$DLT MXV11-B, SBC-11/21

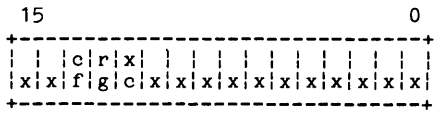
The bit settings of DP.RPS and DP.XPS are used to report the setting of status control bits in the RCSR and XCSR of the specified interface unit. The format of the status-setting word is as follows:



Proceeding from right to left in the preceding format:

- The mc bit (5), if set, indicates that modem control is enabled on the line.
- The sr bit (10), if set, indicates secondary receive.
- The cd bit (12), if set, indicates carrier detect.
- The cs bit (13), if set, indicates clear to send.
- The ri bit (14), if set, indicates that a ring has occurred.

The format of the software status-setting words is as follows:

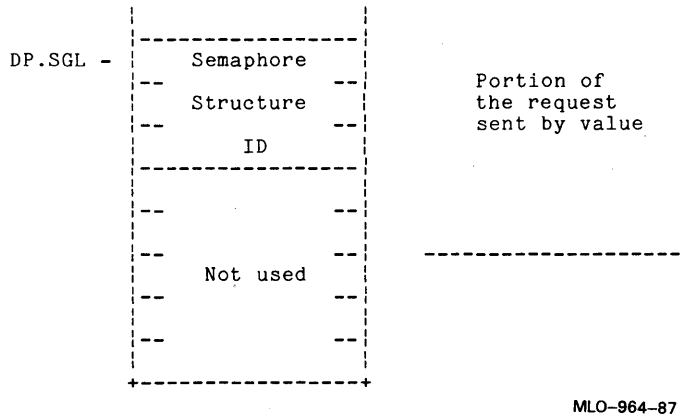


Proceeding from right to left in the format above:

- The xc bit (11), if set, indicates XON/XOFF processing for the input side. No meaning is assigned to the output side.
- The rg bit (12), if set, indicates that the port is connected to a ring buffer. This is a read-only bit.
- The cf bit (13), if set, indicates that the port was connected to a ring buffer during configuration.

C.1.2.6 Report Data-Set Status Change Function

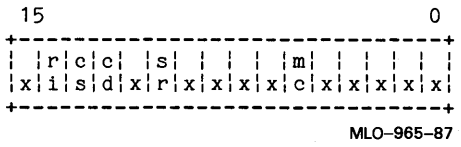
The function-dependent portion of a Report Data-Set Status Change request (function code IF$RSC) is shown below:



The unit number in the function-independent portion of the request selects the desired line-interface unit. Field DP.SGL specifies, by structure ID, the queue semaphore to be signaled when a status change occurs on the specified unit.

The function-dependent portion of a reply to the Report Data-Set Status Change request is the same as a reply to the Get Status function.

The bit settings of word DP.RPS are used to indicate the setting of status control bits in the RCSR of the specified interface unit when a modem-state change occurs. The format of the status-return word (identical to the word returned by Get Status) is as follows:



Proceeding from right to left in the format above:

- The mc bit (5), if set, indicates that modem control is enabled on the line.
- The sr bit (10), if set, indicates secondary receive.
- The cd bit (12), if set, indicates carrier detect.
- The cs bit (13), if set, indicates clear to send.
- The ri bit (14), if set, indicates that a ring has occurred.

C.1.3 Status Codes

The XL driver returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ES$NOR	Normal success
ES$IFN	Invalid function code
ES$NXU	Nonexistent unit
ES$OVR	Overrun error on received data
ES$PAR	Parity error on received data

C.1.4 PDP-11 XL Prefix File

Figure C-1 shows the PDP-11 XL driver prefix module, XLPFX.MAC. There are three other XL prefix files: XLPFXD.MAC, for building an LSI application with debug support; XLPFXF.MAC, for building a FALCON or FALCON-PLUS application with debug support; and XLPFXK.MAC, for building a KXT11-CA application with debug support. Section C.2.4 discusses XLPFXK.MAC. The other three prefix files differ primarily in the default CSR and vector addresses they provide. The default CSR and vector assignments are as follows:

- In XLPFX.MAC, CSR=177560 and VEC=60 (LSI console terminal port)
- In XLPFXD.MAC, CSR=176500 and VEC=300 (Nonconsole terminal port)
- In XLPFXF.MAC, CSR=176540 and VEC=120 (FALCON or FALCON-PLUS SLU2 port)

The following paragraphs describe the prefix file macro calls and symbol definitions that can be edited to fit your application.

$XLPRM labels a word specifying the number of serial line units to be supported by the XL driver. You must modify that value to reflect the number of lines described in the prefix module, that is, the number of line-definition (LINDF$) macros used in the file.

Note

When building an application with debug support, you must not specify a LINDF$ macro for the host-to-target serial line used by PASDBG. That line, which must be connected to the target's console-terminal port, is handled directly by the debug service module. (The default line definition in the XLPFX.MAC file assumes an LSI application without debug support; it describes a line connected to the LSI console terminal port—CSR 177560.)

LINDF$ is a macro call that supplies information about each supported line. One LINDF$ macro must be used per serial line and must supply at least the vector and CSR addresses (receive side only) for the line. A line type must also be specified for other than the basic DLV11 type of line interface unit; the symbol TT$DLE must be specified, for example, for a DLV11–E interface unit. An initial line speed must be specified if a line is configured for programmable baud rate. The other LINDF$ arguments pertain to optional start-up time creation and connection of input and output ring buffers for a given line.

Unit number assignments correspond to the order in which the LINDF$ macros occur in the prefix file. That is, the first or only LINDF$ macro implicitly defines unit 0, the second defines unit 1, the third defines unit 2, and so on.

The complete LINDF$ macro keyword syntax is as follows:

```
LINDF$   vec,csr,typ[=TT$DL],rnam,rsiz[=12.],ratt,rmod,xnam,
xsiz[=80.],xatt,xmod,spd
```

vec

The receive-side interrupt vector address for a given line; the transmit vector for that line is assumed to follow the receive vector by 5 bytes. For example, vec=300 specifies a receive vector at location 300 and implies a corresponding transmit vector at location 304.

csr

The receive-side CSR (RCSR) address for a given line; the transmit CSR (XCSR) for that line is assumed to follow the RCSR by four bytes. For example, csr=176500 specifies an RCSR at location 176500 and implies a corresponding XCSR at location 176504.

typ

The set of functional capabilities to be supported by the line, in terms of an interface unit type:

TT$DL for the minimum common functions provided by a DLV11 or DLV11–J interface

TT$DLE for a DLV11–E interface

TT$DLF for a DLV11–F interface

TT$DLT for a DLART-type interface jumpered for programmable baud rate.

Additional information on type TT$DLT is given below. The default is typ=TT$DL. Note that type TT$DL can be specified for any kind of interface unit, provided that only common DLV11 line functions are utilized.

rnam

The name of a receive ring buffer to be created and automatically connected to the line by the XL driver at start-up time. The name must contain six characters, including trailing blanks, if needed—for example, rnam= <INBUF > . The rnam parameter is optional.

rsiz

The size, in an even number of bytes, of the receive buffer. The default size is 12 bytes. The rsiz parameter is meaningful only if you specify rnam.

ratt

The access attribute of the receive buffer. Specify ratt=SA$RIR for record-oriented input access (default) or ratt=SA$RIS for stream-oriented input access. The ratt parameter is meaningful only if you specify rnam.

rmod

Receive-side mode bits. Specify rmod=F$XCHK to enable automatic XON/XOFF checking.

xnam

The name of a transmit ring buffer to be created and automatically connected to the line by the XL driver at start-up time. The name must contain six characters, including trailing blanks if needed—for example, xnam= <OUTBUF> . The xnam parameter is optional.

xsiz

The size, in an even number of bytes, of the transmit buffer. The default size is 80 bytes. The xsiz parameter is meaningful only if you specify xnam.

xatt

The access attribute of the transmit buffer. Specify ratt=SA$ROR for record-oriented output access (default) or ratt=SA$ROS for stream-oriented output access.

xmod

Transmit-side mode bits; currently unused.

spd

The initial, or start-up, speed setting for a line configured for programmable baud rate. For example, spd=1200. specifies an initial line speed of 1200 baud.

The spd argument sets the receive-side baud rate in the XL driver for the serial line at start-up time, assuming that the line speed is programmable. The spd argument is valid only if the type value is TT$DLE, TT$DLF, or TT$DLT. If the line type is TT$DLE or TT$DLF, the valid speed values are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, or 19,200. (The value 134 indicates 134.5 baud.) If the line type is TT$DLT, the valid speed values are 300, 600, 1200, 2400, 4800, 9600, 19,200, or 38,400.

The optional ring buffer connection capability is provided primarily to support ring buffer I/O via Pascal file variables. It allows the line that will be opened for ring buffer I/O to be connected to named ring buffers by the XL driver at start-up time.

The type symbol TT$DLT indicates a serial line of the type implemented on the MXV11-B multifunction board and on the FALCON or FALCON-PLUS SBC-11/21 microcomputer. That type of line supports programmable baud rate in addition to common DLV11 capabilities. For an MXV11-B line configured for programmable baud rate, or for an SBC-11/21 serial line, specify typ=TT$DLT and give the spd argument. To use a baud rate that has already been set—by the kernel, as specified in the KXT11 configuration macro, for example—omit the spd argument. For an MXV11-B line with hardwired baud rate, indicate the line type as TT$DL.

The XL$xPR definitions specify software priorities associated with the driver process and the hardware priority for all serial line interrupts.

Note that the interrupt vectors specified and implied in the XLPFX.MAC prefix file must also be specified in the system configuration file, using the DEVICES macro.

Figure C–1: XL Driver Prefix File (XLPFX.MAC)

```
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION. ALL
;   RIGHTS RESERVED.
;

   .mcall   MACDF$,IODF$,QUEDF$,DRVDF$,XLISZ$,LINDF$

   macdf$
   quedf$
   drvdf$
   xlisz$   GLOBAL

   dfalt$   F$XCHK,4000

;+
;
; $XLPRM
;
; This table serves as a configuration data area for the XL driver.
; The first word contains the total number of DLV-11 type lines in the
; configuration.  Subsequent data is set by the LINDF$ macro.
; There must be one LINDF$ macro call for each line.
; The csr and vector for each lines receive side must be defined;
; The transmit side csr and vector addresses are assumed to follow
; the receive addresses by 4 bytes each.
;
; The TYP argument specifies a particular type of DL-11.  This is the
; value returned for a get characteristics.  The standard terminal type
; codes are shown below:
;
;   TT$DL      The device supports the minimum common DLV-11
;         type functions
;   TT$DLE      The device supports DLV-11E capabilities
;   TT$DLF      The device supports DLV-11f capabilities
;   TT$DLT      The device supports a DLART, ie. compatible with
;         FALCON, MXV11--B
;
; As an option, ring buffers may be pre-allocated for each line.
; In this case, the driver will create named ring buffer structures
; of a given size and attributes.  The receive and transmit rings
; are defined separately. X-OFF/X-ON checking may be enabled on a
; receive ring buffer if desired with the RMOD parameter.
;
; Currently, unit numbers for each line correspond to the order in which
; the LINDF$ macros are called.
;
; For example:
;
; $XLPRM:: .word   2      ; Define two lines
;
;   LINDF$   csr=177560,vec=60,typ=TT$DLF,rnam=<XLI0  >,rsiz=10.,
;      ratt=SA$RIS,rmod=F$XCHK,xnam=<XL00  >,xsiz=80.,xatt=SA$ROS
;
;   LINDF$   csr=176500,vec=300
;
;    .end
```

```
;
; Defines a line on unit 0 with predefined stream-attribute buffers for
; both receive and transmit sides and defines another line on unit 1.
; Unit 0 above has X-OFF/X-ON checking enabled.  The controller for unit 0
; is a DL-11F.  Note that you must pass a 6 character blank padded string
; for the ring buffer structure names as shown above.
;
; The data defined by this macro is used by the XL drivers initialization
; routines to connect to interrupt vectors and create ring buffer
; structures.
;
;-
   .GLOBL   $XL    ;Haul in the XL driver from the library

XL$PPR   ==   175.        ; Process priority
XL$FPR   ==   175.*256.   ; Fork process priority
XL$HPR   ==   4           ; hardware priority
XL$IPR   ==   250.        ; process initialization priority

   pdat$
$XLPRM::.word   1    ; Define only one line

LINDF$   csr=177560,vec=60,rmod=F$XCHK,rnam=<XLI0  >,xnam=<XL00  >
   .end
```

C.2 Peripheral Processor XL Driver

The peripheral processor XL device driver supports asynchronous I/O operations on devices connected to any of the three serial I/O ports on the peripheral processor. This driver is identical to the PDP–11 XL driver described in Section C.1, except that it includes support for the multiprotocol chip that resides on the peripheral processor. Thus, the driver can concurrently service up to three serial ports on the peripheral processor.

Note

The three serial ports on the peripheral processor can be used by the TU58 (DD) driver as well as by the XL driver. See Chapter 4 for a description of the DD driver interface.

In addition, one of the peripheral processor's serial lines—the multiprotocol chip "A" port—can be used for synchronous serial I/O via the XS device driver. See Chapter 13 for a description of the XS driver interface.

The first serial I/O port on the peripheral processor is a standard DL Asynchronous Receive/Transmit (DLART) device. The second port provides all the features of a DLV11–E, including modem control, but has a different hardware interface. The third port provides all the features of a standard DLART device but has a different hardware interface.

The driver performs serial input/output operations in either block mode or ring mode. In block-mode input, a specified number of bytes are transferred directly from the serial line unit to the requester's buffer space. In block-mode output, a specified number of bytes are transferred from the requester's data buffer to the serial line unit.

In ring-mode input, the driver continuously transmits bytes from the serial line unit to an input ring buffer specified by the requester. Once initiated by a Connect Receive Ring Buffer request for a given unit, the input operation continues until the ring buffer is disconnected.

In ring-mode output, the driver continuously transmits bytes from an output ring buffer, as they become available, to the specified serial line unit. The requester or some other process must first put characters into the ring buffer and then issue a Connect Transmit Ring Buffer request, which starts the output transfer. The output operation continues until the ring buffer is disconnected. Ring mode is intended for unbounded or interactive data paths.

For either type of output request, the driver provides optional XON/XOFF control character processing. If requested to do so, the driver inhibits the output side of a given serial line unit when it detects an XOFF character on the input side of the same channel. Output resumes when an XON is subsequently received.

For the port that supports modem control, the driver allows you to enable data-set interrupts and to receive data-set status information when such interrupts occur. The driver allows you to get data-set status information and to set certain XCSR and RCSR bits. By this mechanism, you can control baud rates, enable interrupts if a modem control signal changes, set modem control signals, and transmit break signals.

The driver's request queue semaphore name is $XLA. The desired serial line is specified in the function request by a configuration-determined unit number.

The configuration for all asynchronous serial devices attached to a processor must be specified in the peripheral processor XL driver prefix file XLPFXK.MAC. (See Section C.2.4.)

C.2.1 Functions Provided

The functions provided by the peripheral processor XL driver are listed below by symbolic and decimal function code:

Code	Function Performed
IF$RDP (0)	Read Physical
IF$RDL (1)	Read Logical (equivalent to Read Physical)
IF$WTP (3)	Write Physical
IF$WTL (4)	Write Logical (equivalent to Write Physical)
IF$SET (6)	Set Status
IF$GET (7)	Get Status
IF$CRR (8)	Connect Receive Ring Buffer
IF$CXR (9)	Connect Transmit Ring Buffer
IF$DRR (10)	Disconnect Receive Ring Buffer
IF$DXR (11)	Disconnect Transmit Ring Buffer
IF$RSC (12)	Report Data-Set Status Change

C.2.1.1 Read Function

The read function (code IF$RDP or IF$RDL) is performed in block mode. The driver reads a specified number of characters from the serial line unit into the buffer area identified in field DP.BUF of the request message.

Read requests to a unit that has been connected to a ring buffer are not supported.

C.2.1.2 Write Function

The write function (code IF$WTP or IF$WTL) is performed in block mode. The source and amount of the data to be transferred are specified by the DP.BUF and DP.LEN fields of the request message.

The function modifier bit FM$XCK (bit 11) in the function code word (offset DP.FUN) is used to enable or disable automatic XON/XOFF processing. If the bit is set, the driver intercepts any XOFF character from the input side of the line and disables output for the same channel until a subsequent XON character is received. Multiple successive XON or XOFF characters have the same effect as just one such character. If XON/XOFF processing is not selected, the driver passes all input characters to the requester. The XON/XOFF function modifier bit has no effect if the feature has been permanently enabled via a Set Status request.

Write requests to a unit that has been connected to a ring buffer are not supported.

C.2.1.3 Connect Receive Ring Buffer Function

The Connect Receive Ring Buffer function (code IF$CRR) connects a user-specified ring buffer to a serial line unit and initiates input to that ring buffer. Any input occurring on the line is transferred to the ring buffer identified in request field DP.SGL; field DP.BUF of the request is ignored. Input into the ring buffer continues until the ring buffer is disconnected.

Note

When a line is connected to a ring buffer, all hardware exceptions are ignored, including parity exceptions, framing exceptions, and overrun exceptions.

C.2.1.4 Disconnect Receive Ring Buffer Function

The Disconnect Receive Ring Buffer function (code IF$DRP) disconnects a user-specified ring buffer from a serial line unit. Any subsequent input occurring on the line is ignored.

This request will not disconnect a ring buffer that was attached to a line by the driver prefix file.

C.2.1.5 Connect Transmit Ring Buffer Function

The Connect Transmit Ring Buffer function (code IF$CXR) connects a user-specified ring buffer to a serial line unit and initiates output from that ring buffer. Any data put into the ring buffer identified in request field DP.SGL is output on the specified line. Field DP.BUF is not used. The function modifier bit FM$XCK is used to enable or disable automatic XON/XOFF processing as previously described in the write function section.

C.2.1.6 Disconnect Transmit Ring Buffer Function

The Disconnect Transmit Ring Buffer function (code IF$DXR) disconnects a user-specified ring buffer from a serial line unit.

C.2.1.7 Set Status Function

The Set Status function (code IF$SET) sets the specified unit's hardware transmit control register (XCSR) and/or the receive control register (RCSR) according to values that are contained in the request. (For the multiprotocol chip, the hardware is not formatted in that manner, but the information is unpacked from the 2-word format so that applications written for a DL-type device will run on a multiprotocol device.) The receiver software status word in the request allows the permanent enabling of XON/XOFF processing or allows XON/XOFF processing to be selected via the function modifier bit in the write function.

C.2.1.8 Get Status Function

The Get Status function (code IF$GET) returns a packet containing the class and type of hardware, the software status of the receive and transmit circuits, and two words read from the hardware. (For the multiprotocol chip, the hardware is not formatted in that manner, but the information is packed into a 2-word format so that applications written for DL-type devices will run on a multiprotocol device.)

C.2.1.9 Report data-set status change function

The Report Data-Set Status Change function (code IF$RSC) allows the requester to wait for a change of data-set (modem) status. When a change occurs, the driver returns a standard Get Status reply, as described in the Get Status sections. When this function is requested, the driver assumes that modem control has been enabled via a previous Set Status (IF$SET) request. If modem control was not enabled, the wait process will hang indefinitely.

For this function, you must specify a queue semaphore at offset DP.SGL that is to be signaled when a status change occurs on the specified unit. For each change request received, a Get Status reply is returned immediately, and a second reply in the same format is returned when a change occurs in the state of the modem inputs. Although several requests can be sent at once, only one reply will be returned per input change.

C.2.1.10 Device-Independent Function Modifiers

If modifier FM$BSM (bit 13) of DP.FUN is set, the peripheral processor XL driver signals a binary or counting semaphore, as described in Chapter 1.

C.2.2 Function-Dependent Request Formats

The function-independent portion of a driver request message is described in Chapter 1.

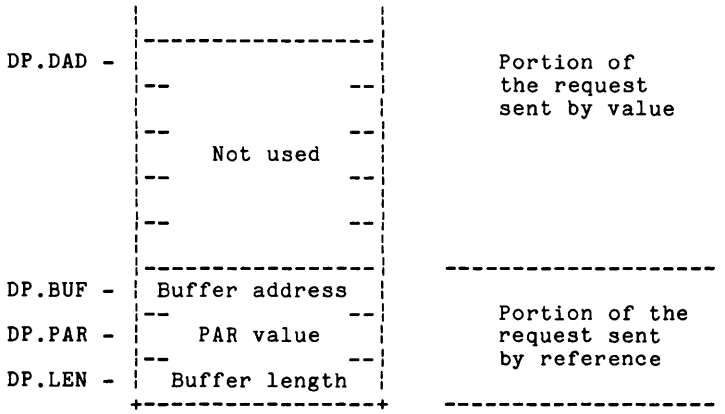
Note

For the Disconnect Transmit Ring Buffer function (code IF$DXR), field DP.ALN in the function-independent portion of the reply packet returns the number of untransmitted bytes that remain in the disconnected ring buffer.

The function-dependent portion of an XL driver request following field DP.SEM is described below for each type of function.

C.2.2.1 Block-Mode Read or Write Functions

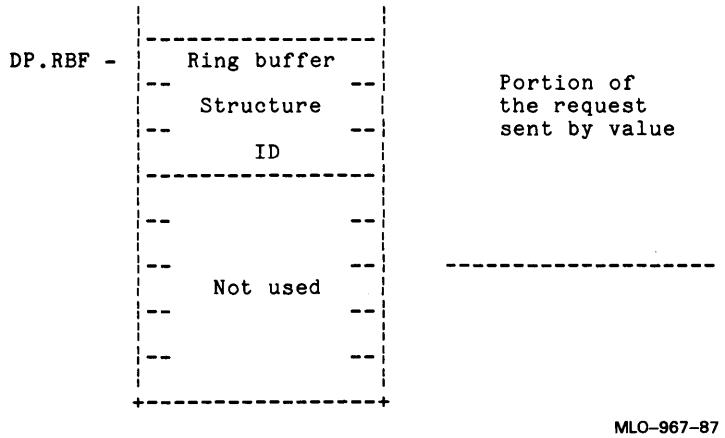
The function-dependent portion of a block-mode read or write request is shown below:



The unit number in the function-independent portion of the request selects the desired serial unit; unit numbering starting at 0. The buffer address specifies the destination of the data to be read or the source of the data to be written. The buffer-length value determines the length, in bytes, of the data transfer. The PAR value is filled in by the operating system.

C.2.2.2 Connect Receive or Transmit Ring Buffer Functions

The function-dependent portion of a Ring Buffer Connect request for either input or output is shown below:

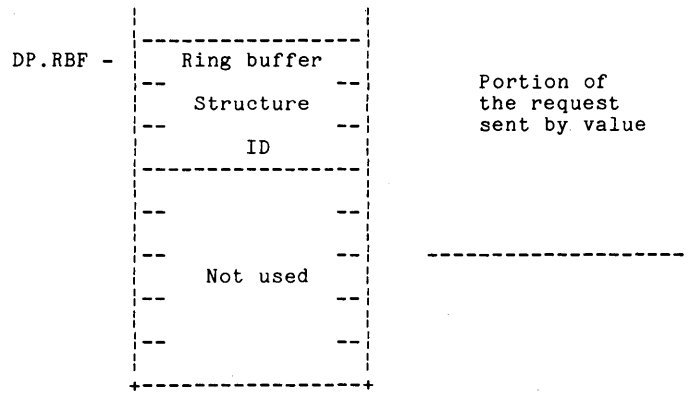


The unit number in the function-independent portion of the request selects the desired serial unit; unit numbering starts at 0. Field DP.RBF specifies, by structure ID, the destination ring buffer for an input operation or the source buffer for an output operation. On input, individual characters are put into the ring buffer as they are received; on output, individual characters are

transmitted from the ring buffer as they become available by action of the user process. In either case, the length of the transfer is unlimited.

C.2.2.3 Disconnect Receive or Transmit Ring Buffer Functions

The function-dependent portion of a Ring Buffer Disconnect request for either input or output is shown below:

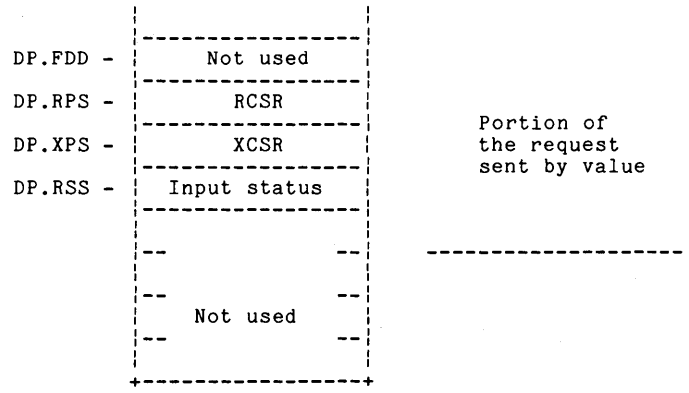


MLO-968-87

The unit number in the function-independent portion of the request selects the serial unit from which the ring buffer is to be disconnected. Field DP.RBF specifies, by structure ID, the ring buffer that was previously connected to the unit in question. All input to or output from the ring buffer ceases when the request is acted on by the driver.

C.2.2.4 Set Status Function

The function-dependent portion of a Set Status request (code IF$SET) is shown below:

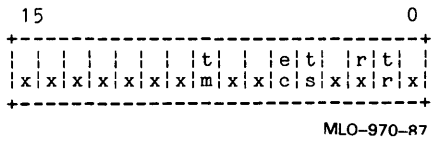


MLO-969-87

The previous request packet fields have the following significance:

Field	Significance
DP.RPS	Status control bits to be set in the multiprotocol receiver control hardware (equivalent to DL-device receiver CSR); not used for the DLART device
DP.XPS	Status control bits to be set in the DLART transmitter CSR or in the multiprotocol hardware equivalent; the bit settings are hardware-dependent
DP.RSS	Receiver software status bit settings

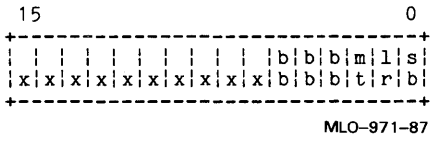
The format of the receiver status-setting word (offset DP.RPS) is as follows:



Proceeding from right to left in the format above:

- The tr bit (1), if set, indicates data terminal ready.
- The rx bit (2), if set, indicates request to send.
- The ts bit (4), if set, indicates terminal in service.
- The ec bit (5), if set, enables modem control (modem control lines are active).
- The tm bit (8), if set, indicates test mode.

The format of the transmitter status-setting word (offset DP.XPS) for the DLART device is as follows:



Proceeding from right to left in the format above:

- The sb bit (0), if set, requests a BREAK to be transmitted on the output line.
- The lr bit (1), if set, allows a new baud rate to be loaded (bit PBRE on hardware).
- The mt bit (2), if set, enables self-test; whatever is written to the transmit data register is looped back and received by the receive data register.
- The bb bits (3 to 5) indicate the desired baud rate for the unit (bits PBR0, PBR1, and PBR2 on hardware).

The format of the transmitter status-setting word (offset DP.XPS) for a multiprotocol device is as follows:

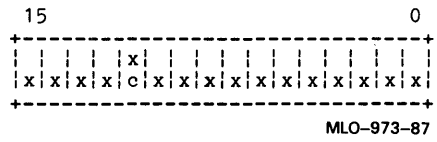
MLO-972-87

Proceeding from right to left in the format above:

- The sb bit (0), if set, requests a BREAK to be transmitted on the output line; the bit must be cleared when the baud rate is set.
- The bb bits (1 to 15) indicate the desired baud rate; the valid octal values are:

Baud Rate	Value
307.2Kb	2
153.6Kb	4
76.8Kb	10
38.4Kb	20
19.2Kb	40
9600	100
4800	200
2400	400
1200	1000
600	2000
300	4000
150	10000
110	12722

The format of the software status-setting word (offset DP.RSS) is as follows:

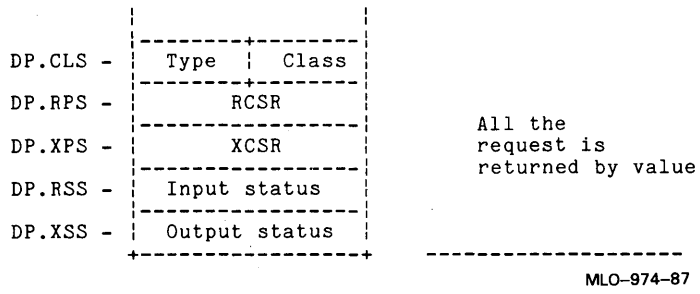


MLO-973-87

The xc bit (11), if set, enables XON/XOFF processing regardless of the state of the XON/XOFF function modifier bit within the block write or Connect Transmit Ring Buffer commands. (See Section C.2.2.1.)

C.2.2.5 Get Status Function

The function-dependent portion of a reply to a Get Status request (function code IF$GET) is shown below:



In the information above:

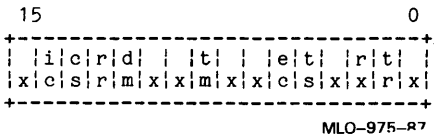
- Class is DC$TER, indicating a serial line.
- Type indicates the specific type of serial line device, as follows:

TT$DLT	Serial line with programmable baud rate, exception indicator flags, and a self-test mode—the DLART port on the KXT11-CA.
TT$DM	First or second port on the multiprotocol chip on the KXT11-CA when used in the asynchronous data-leads-only mode. The port then supplies the functionality of the TT$DLT type device.
TT$DMM	Multiprotocol line operating in asynchronous serial mode. Contains full modem control, mandatory programmable baud rate, mandatory programmable vector address, and exception indicator flags. Only the first port on the multiprotocol device can be this device type and then only when this driver or no driver is in control of the second multiprotocol port.

The other reply packet fields shown above have the following significance:

Field	Significance
DP.RPS	Status control bits returned from the multiprotocol receive control hardware (equivalent to DL-device receiver CSR) if modem control is in use; not used for the DLART device
DP.XPS	Status control bits returned from the DLART transmitter CSR or from the multiprotocol hardware equivalent; the bit settings are hardware-dependent
DP.RSS	Receiver software status bit settings
DP.XSS	Transmitter software status bit settings

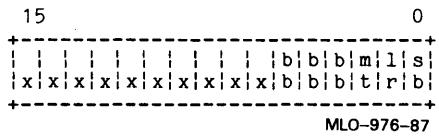
The format of the receiver status word (offset DP.RPS) is as follows:



Proceeding from right to left in the format above:

- The tr bit (1), if set, indicates data terminal ready.
- The rx bit (2), if set, indicates request to send.
- The ts bit (4), if set, indicates terminal in service.
- The ec bit (5), if set, indicates that modem control is enabled on the line; modem control lines are active.
- The tm bit (8), if set, indicates test mode.
- The dm bit (11), if set, indicates data mode (TT$DMM type only).
- The rr bit (12), if set, indicates receiver ready.
- The cs bit (13), if set, indicates clear to send.
- The ic bit (14), if set, indicates incoming call (TT$DMM type only).

The format of the transmitter status word (offset DP.XPS) for the DLART device is as follows:



Proceeding from right to left in the format above:

- The sb bit (0), if set, indicates transmitting break.
- The lr bit (1), if set, indicates programmable baud rate enabled (bit PBRE on hardware).
- The mt bit (2), if set, indicates that self-test mode is active; whatever is written to the transmit data register is looped back and received by the receive data register.
- The bb bits (3 to 5) indicate the baud rate for the unit (bits PBR0, PBR1, and PBR2 on hardware).

The format of the transmitter status word (offset DP.XPS) for a multiprotocol device is as follows:

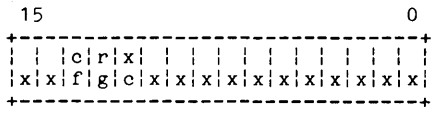
MLO–977–87

Proceeding from right to left in the format above:

- The sb bit (0), if set, indicates transmitting BREAK.
- The bb bits (1 to 15) indicate the baud rate; the octal values are:

Baud Rate	Value
307.2Kb	2
153.6Kb	4
76.8Kb	10
38.4Kb	20
19.2Kb	40
9600	100
4800	200
2400	400
1200	1000
600	2000
300	4000
150	10000
110	12722

The formats of the receiver and transmitter software status words (offsets DP.RSS and DP.XSS) are identical, as follows:



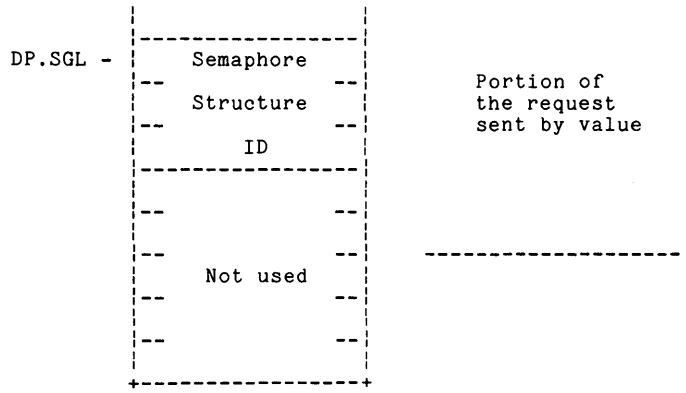
MLO–978–87

Proceeding from right to left in the format above:

- The xc bit (11), if set, indicates XON/XOFF processing for the input side. No meaning is assigned to the output side.
- The rg bit (12), if set, indicates that the port is connected to a ring buffer.
- The cf bit (13), if set, indicates that the port was connected to a ring buffer during configuration.

C.2.2.6 Report Data-Set Status Change Function

The function-dependent portion of a Report Data-Set Status Change request (function code IF$RSC) is shown below:



The unit number in the function-independent portion of the request selects the desired serial line. Field DP.SGL specifies, by structure ID, the queue semaphore to be signaled when a status change occurs on the specified unit.

The function-dependent portion of a reply to the Report Data-Set Status Change request is the same as a reply to the Get Status function.

C.2.3 Status Codes

The XL driver returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ES$NOR	Normal success
ES$IFN	Invalid function code
ES$NXU	Nonexistent unit
ES$OVR	Overrun exception on received data
ES$PAR	Parity exception on received data

C.2.4 KXT11-CA XL Prefix File

The XL prefix module to configure the XL driver for a KXT11-CA, XLPFXK.MAC, is similar to the XL prefix files XLPFX.MAC, XLPFXD.MAC, and XLPFXF.MAC. (See Section C.1.4.) However, because the configuration of the board is fixed, fewer modifications to the file are required. There are always three serial lines: the console line at vector 60, multiprotocol channel A, and multiprotocol channel B. You should normally use three LINDF$ macros to configure the XL driver, as illustrated in Figure C–2 and described below. However, if you connect a TU58 to

the KXT11–CA, the TU58 driver will use one of the serial lines. If this is the case, be sure that XLPFXK.MAC does not also define the same line for the XL driver. If you use the XS driver, the XS driver will always use multiprotocol channel A; in that case, be sure to omit channel A from the XL driver configuration.

typ

The following typ values are permitted in the LINDF$ macro for the XL driver with the KXT11–CA:

- TT$DLT for the console channel at vector 60
- TT$DM for KXT11–CA multiprotocol channel data only
- TT$DMM for KXT11–CA multiprotocol channel with full modem control. Multiprotocol channel A is the only channel that can be of this type, and then only when the XL driver (or no driver) is in control of multiprotocol channel B.

vec

The vectors for the asynchronous lines on the KXT11–CA are:

Line	Vector
Console	60
Channel A	140
Channel B	160

csr

The control status registers for the three lines are:

Line	CSR
Console	177560
Channel A	175700
Channel B	175710

The other LINDF$ arguments correspond to the LINDF$ arguments in XLPFX.MAC, described in Section C.1.4.

Figure C–2: KXT11–CA XL Driver Prefix File (XLPFXK.MAC)

```
.title   XLPFXK   KXT11-CA XL DEVICE DRIVER PREFIX MODULE

;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1986 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.
;

   .mcall   MACDF$,IODF$,QUEDF$,DRVDF$,XLISZ$,LINDF$

   macdf$
   quedf$
   drvdf$
   xlisz$   GLOBAL

   dfalt$   F$XCHK,4000

;+
;
;   This module contains an example of using the configuration macros
;  to configure asynchronous serial lines on the KXT11-CA.
;
; $XLPRM
;
; This table serves as a configuration data area for the XL driver.
; The first word contains the total number of asyncronous serial lines
; in the configuration.  Subsequent data is set by the LINDF$ macro.
; There must be one LINDF$ macro call for each line.
; The csr and vector for each line's receive side must be defined;
; The transmit side csr and vector addresses are assumed to follow
; the receive addresses by 4 bytes each.
;
; The TYP argument specifies a particular type of Serial device. This is
; the value returned for a get characteristics.  The standard terminal type
; codes are shown below:
;
;   TT$DL       The device supports the minimum common DLV-11
;               type functions
;   TT$DLE      The device supports DLV-11E capabilities
;   TT$DLF      The device supports DLV-11f capabilities
;   TT$DLT      The device supports a DLART, ie. compatible with
;               FALCON, MXV11--B, or the console port on the KXT11-CA.
;   TT$DM       The data leads only multiprotocol channel on KXT11-CA
;   TT$DMM      The multiprotocol channel with modem control on the
;               KXT11-CA.
;
; As an option, ring buffers may be pre-allocated for each line.
; In this case, the driver will create named ring buffer structures
; of a given size and attributes.  The receive and transmit rings
; are defined separately. X-OFF/X-ON checking may be enabled on a receive
; ring buffer if desired with the RMOD parameter.
;
; Currently, unit numbers for each line correspond to the order in which
; the LINDF$ macros are called.
;
```

```
; For example:
;
; $XLPRM:: .word   2       ; Define two lines
;
;   LINDF$   csr=177560,vec=60,typ=TT$DLT,rnam=<XLI0  >,rsiz=10.,
;   ratt=SA$RIS,rmod=F$XCHK,xnam=<XL00>,xsiz=80.,xatt=SA$ROS,spd=9600
;
;   LINDF$   typ=TT$DM,csr=175710,vec=160,spd=9600
;
;   .end
;
; Defines a line on unit 0 with predefined stream-attribute buffers for
; both receive and transmit sides and defines another line on unit 1.
; Unit 0 above has X-OFF/X-ON checking enabled.  The controller for unit 0
; is a DLART.  Note that you must pass a 6 character blank padded string
; for the ring buffer structure names as shown above.
;
; The data defined by this macro is used by the XL drivers initialization
; routines to connect to interrupt vectors and create ring buffer structures.
;
;-

       .GLOBL   $XL

XL$PPR  ==      175.      ; Process priority
XL$HPR  ==      4         ; hardware priority for port on KXT11-CA
XL$IPR  ==      250.      ; Process initialization priority
       pdat$
$XLPRM::.word   2         ; Define two lines

;    Multiprotocol channel B with ring buffers and data leads only
;
LINDF$   typ=TT$DM,csr=175710,vec=160,rmod=F$XCHK,rnam=<XLI0>,rsiz=134.,
         xnam=<XL00  >,xsiz=80.,spd=9600

;    Multiprotocol channel A with ring buffers and modem control
;
LINDF$   typ=TT$DMM,csr=175700,vec=140,rmod=F$XCHK,rnam=<XLI1>,rsiz=134.,
         xnam=<XL01  >,xsiz=80.,spd=9600
        .end
```

Appendix D

Sample MACRO–11 Device Driver

Source code for the RX02 device driver, DYDRV.MAC, is provided in this appendix as a sample device driver written in MACRO–11. The associated prefix file, DYPFX.MAC, is listed in Chapter 4 and is available on the distribution kit.

The DY driver's impure-area definition macro, DYISZ$, resides in the COMU and COMM macro libraries and is invoked in the driver source code. See Section 14.3 for a listing of the DYISZ$ macro.

In this appendix, long macro invocations (DFSPC$ and CRPC$) are continued on a second line. When writing source code in MACRO–11, however, the entire invocation must be on one line.

Note

For an example of a driver coded in Pascal, see the DRV11 (YA) driver source files (YADRV.PAS and YAPFX.PAS) included on the distribution kit. The YA driver is described in Chapter 6.

```
        .nlist          ;Edit Level 4
        .enabl  LC
        .list
        .title  DYDRV - RX02 Driver
        .ident  /V02.00/
;
;               COPYRIGHT (c) 1982, 1986 BY
;          DIGITAL EQUIPMENT CORPORATION, MAYNARD
;           MASSACHUSETTS.  ALL RIGHTS RESERVED.
;
```

```
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
; ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
; COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
; OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
; TRANSFERRED.
;
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
; AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
; CORPORATION.
;
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
; SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;

        .SBTTL    Edit History
;+
;
; Module name: DYDRV.MAC
;
; System: MicroPower/Pascal
;
; Author: ERS             Creation date: 21-JUL-82
; ERS                     15-NOV-82 fix IS$OFL to IS$OVF
; ERS                     20-APR-83 fix recursive errors
; ERS                     24-Feb-84 Update for V2.0
;
;
; Functional Description:
;
;       This module provides device driver services for the RX02 flexible
;       diskette. It controls any number of RX02 drives.  The driver can
;       read, write, and format single or double density diskettes.
;
;-

       .sbttl   Declarations
;+
;
; System macro requirements
;
;-
        .enabl  GBL

        .mcall  macdf$, iodf$, quedf$, drvdf$, dcdf$, ccdf$, dyisz$

        macdf$
        iodf$
        quedf$
        drvdf$
        dcdf$
        ccdf$
        dyisz$

       .mcall  cint$s, crpc$p, crst$s, dapk$s, dint$s, dlpc$,  dlst$s
       .mcall  fork$,  sgnl$s, sglq$s, waiq$s, wait$s, waqc$s, xtad$
```

```
;+
;
; Local macro definitions:
;
;-

;+
;
; External symbols which are necessary to assemble this module but
; which may have default definitions:
;
;-

       dfalt$   DY$RTY,10.      ; Retry count


;+
;
; External symbols defined in the driver prefix module.
;
;-
   .globl   DY$IPR       ; DY initialization priority
   .globl   DY$PPR       ; DY process priority
   .globl   DY$HPR       ; RX02 hardware priority

;+
;
; Data and symbols owned by this module are defined here and storage is
; allocated in the appropriate data section.
;
;-

; Local symbol definitions:

;+
;
; Define offsets within and size of the impure area required by the DY
; driver for each controller process.
;
;-
```

```
        psect$    .TEMP.,<RO,D,LCL,ABS,OVR>
.temp.:
REQSEM:   .blkb   SD.SIZ          ; Request semaphore SDB
UNTMAP:   .blkb                   ; Unit allocation bit map.  Only 2 bits
                                  ; required since only 2 units per controller
ADTYPE:   .blkb                   ; Physical or logical
                                  ; 0 physical,1 logical
          .even
SIZE:     .blkb                   ; flag for get characteristics
FORMAT:   .blkb                   ; flag for format command
RTYCNT:   .blkb                   ; Total number of retrys (0 if RTYINH)
          .even
SAVDEN:   .blkw   DY$MXU          ; saved density for each drive
DENPTR:   .blkw                   ; -> savden[unit]
SAVR3:    .blkw                   ; saved R3 for returns after interrupt
INTRTN:   .blkw                   ; interrupt return address
PACKET:   .blkw                   ; Pointer to request packet
ERROR:    .blkw                   ; Error code
STATUS:   .blkw                   ; Status code
SECSIZ:   .blkw                   ; sector size in words
LSN:      .blkw                   ; logical sector number (logical accessing)
SECTOR:   .blkw                   ; Sector (only if physical accessing)
TRACK:    .blkw                   ; Track      "
WRDCNT:   .blkw                   ; # of words this request
COUNT:    .blkw                   ; # of words this transfer
TEMP:     .blkw                   ; Temporary word for dyerr
BUFFER:   .blkw                   ; Buffer address of user buffer
RX2CSR:   .blkw                   ; Address of RX02 CSR
RX2DB:    .blkw                   ; Address of RX02 Data Buffer
VECTOR:   .blkw                   ; Interrupt vector
ISRENB:   .blkw                   ; ISR has been enabled  ie. main process
                                  ; is or will shortly block on $DYDNE semaphore
SAVSP:    .blkw                   ; saved stack pointer
$DYDNE:   .blkw                   ; Address of control SDB
ISRIMP:                           ; ISR impure area
CNTSEM:   .blkb   SD.SIZ          ; Control semaphore sdb
          .even
..ISZ     =       . - .temp.      ; Size of impure area.

; Validate parameters specified in the dyisz$ macro
.if LT    <DY$ISZ - ..ISZ>
          .error  DY$ISZ  ; Impure area too small as specified in DYISZ$
          .error  ..ISZ   ; Edit DYISZ$ in DRVDEF, replacing value for DY$ISZ
.endc

.if GT    <DY$MXU - 2>
          .error  DY$MXU  ; RX02 supports a maximum of 2 drives per controller
.endc

.if LT    <DY$SSZ - <$MINST+110>>
          .error  DY$SSZ  ; Stack size specified in DYISZ$ is too small
.endc

.if GT    DY$USZ
          .error  DY$USZ  ; Value specified in DYISZ$ is unnecessary
.endc

; RX02 Device Register Definitions:

; Control and Status Register Bit Definitions
```

```
CSGO    =      1               ; Initiate function

CSFUN   =     16               ; function bits
CSUNIT  =     20               ; Unit bit
CSUN1   =     20               ; Unit one
CSDONE  =     40               ; Done bit
CSINT   =    100               ; Interupt enable
CSTRAN  =    200               ; Transfer request (implies RX02 is
                               ; ready for rest of command string.
CSDN    =    400               ; Double density request
CSHEAD  =   1000               ; Select second head
CSRX02  =   4000               ; Controller is RX02
CSXM    =  30000               ; XM bits
CSINIT  =  40000               ; RX11 initialize
CSERR   = 100000               ; Error

; CSR Function Codes in Bits 1-3

CSFBUF  = 0*2                  ;0 - Fill silo (pre-write)
CSEBUF  = 1*2                  ;1 - Empty silo (post-read)
CSWRT   = 2*2                  ;2 - Write sector
CSRD    = 3*2                  ;3 - Read sector
CSFMT   = 4*2                  ;4 - Format
CSRDST  = 5*2                  ;5 - Read status
CSWRTD  = 6*2                  ;6 - Write sector with deleted data
CSMAIN  = 7*2                  ;7 - Maintenance

assum$   CSRD&2   NE 0         ; 2 bit must be on  in read
assum$   CSWRT&2  EQ 0         ; 2 bit must be off in write
assum$   CSWRTD&2 EQ 0         ; 2 bit must be off in write

; Error and Status Register Bit Definitions

ESCRC   =      1               ; CRC error
ESSID1  =      2               ; Side 1 ready
ESID    =      4               ; Initialize done
ESACLO  =     10               ; RX power failure
ESDNER  =     20               ; Density error
ESDDEN  =     40               ; Drive density
ESDDAT  =    100               ; Deleted data mark
ESDRDY  =    200               ; Drive ready
ESUNIT  =    400               ; Unit selected
ESHEAD  =   1000               ; Head selected
ESWDCT  =   2000               ; Word count overflow during fill or empty
ESNXM   =   4000               ; Non-existent memory

; Device parameters

LBSIZE   =   988.              ; RX02 logical block size
NUMSEC   =   26.               ; Number of sectors
NUMCYL   =   77.               ; Number of cylinders (tracks in hdw parlance)
NUMTRK   =   1                 ; Number of tracks (surfaces)
SECTSZ   =   256.              ; Number of bytes/sector double density


        .sbttl   Process definition

dfspc$   pid=$DYADR,pri=DY$IPR,cxo=0,typ=PT.DRV,grp=1,ter=DYSTP,cxl=0,
  sti=$DYAG2,stl=$DYAG1,sth=$DYAG2,start=DYINIT,ini=0

        pdat$
```

```
$DYPCD::
crpc$p   pdb=PDB,pri=DY$PPR,cxo=0,grp=1,ter=DYSTP,cxl=0,sti=0,stl=0,sth=0,
  start=DYSTRT,ini=0

        impur$

PDB:    .blkb   SD.SIZ          ; PDB for creating driver processes

        pdat$

; Action Routine Dispatch Table:

PLTABL:  .byte   0                      ; Read physical
         .byte   1                      ; Read logical
         .byte   200                    ; Reserved
         .byte   0                      ; Write physical
         .byte   1                      ; Write logical
         .byte   200                    ; Reserved
         .byte   377                    ; Get characteristics
         .byte   377                    ; Set characteristics
         .even

; Device Characteristics Table:

DTABLE:  .byte   DC$DSK                 ; Class is disk
         .byte   DK$DY2                 ; Type is RX02
         .word   LBSIZE,0               ; Number of logical blocks
         .byte   NUMSEC                 ; Number of sectors
         .byte   NUMTRK                 ; Number of tracks
         .word   NUMCYL                 ; Number of cylinders
DTBSIZ   =       . - DTABLE             ; Size of table

; Table of error returns corresponding to bits in the error and status register

ERLIST:
        ES$CTL                  ; Return controller error if no error bits set
        ES$PAR                  ; Parity error for CRC error
        0                       ; Nothing for Side 1 ready
        0                       ; Nothing for initialize done
        ES$PWR                  ; Power fail for RX power failure
        ES$IVM                  ; Invalid mode for density error
        0                       ; Nothing for drive density
        0                       ; Nothing for deleted data mark
        ES$UNS                  ; Unsafe for drive not ready
        0                       ; Nothing for unit selected
        0                       ; Nothing for reserved bit
        ES$OVF                  ; Word count overflow during fill or empty
        ES$NXM                  ; Non-existent memory

        pure$
```

```
       .sbttl   DYINIT   - Power Up Initialization Process
;+
;
; DYINIT - This process creates an instance of the device
; driver process for each controller (after the first, this
; process becomes the first controller process) present on
; the particular configuration. It creates a request semaphore
; (named successively $DYA, $DYB, ...) for each controller
; specified by the $DYCFG table created by the configuration
; procedure.
;
; Control passes to this process on power up. It runs at highest
; priority (255.) and then lowers its priority to become the first
; controller process.
;
;-

DYINIT:   MOV   #$DYPUR,R5    ; -> Configuration data for RX02
   CALL   $DDINI              ; Common device driver initialization
                              ; routine

   .sbttl   DYSTRT   - Request Process Init Procedure
;+
;
; DYSTRT - This is the entry point for each process serving a device
; controller. A pointer to the impure area of the controller
; is passed on the stack.
;
;    (SP)   Pointer to impure area
;
;-

DYSTRT:   MOV   (SP)+,R5            ; -> impure area
          MOV   (SP)+,R4            ; -> initialization data
          BIT   (SP)+,R0            ; Throw away the controller id

; Create an unnamed binary semaphore to serve as an interrupt semaphore
; connecting the ISR to the I/O request service process.

          MOV    R5,R2              ; Copy impure pointer
          ADD    #CNTSEM,R2         ; -> Interrupt semaphore
          CLR    SD.NAM(R2)         ; Clear name field in SDB
          MOV    R2,$DYDNE(R5)      ; Save pointer to SDB

          assum$ ST.BSM eq 0
;                              #ST.BSM
          crst$s sdb=R2, styp=#0, satr=#0, value=#0
          BCS    20$                 ; If CS, create semaphore failed

          MOV    CC.PCS(R4),R1       ; -> CSR/VECTOR LIST
          MOV    (R1),RX2CSR(R5)     ; -> CSR
          MOV    (R1)+,RX2DB(R5)     ; Compute -> Data buffer register
          ADD    #2,RX2DB(R5)        ;       "
          MOV    (R1),VECTOR(R5)     ; -> VECTOR

; Connect the interrupt vector to the ISR.

   cint$s vec=(R1),ps=#DY$HPR*40,val=#0,isr=#$DYINT,imp=R5,pic=#0
          BCC    30$                 ; Br if no error
```

```
20$:    CALL    $DDEXC              ; report kernel error (code in R0)
                                    ; no return

30$:     MOV    SP,SAVSP(R5)        ; save the stack pointer
         push$  #DY$MXU             ; Maximum number of units on RX02
         push$  R4                  ; Compute address of unit list
         ADD    #CC.USP,(SP)        ;   "
         push$  R5                  ; Compute address of unit bitmap
         ADD    #UNTMAP,(SP)        ;   "
         CALL   $UNTMP              ; Set bitmap for supported drives
         TST    (SP)+               ; Any errors?
         BNE    DYREQ               ; Br if not
         MOV    #ES$ICD,R0          ; Invalid config data
         BR     20$                 ; report error

   .sbttl   DYREQ   - Request Process Queue Server
;+
;
; This is the I/O request service process. This process
; starts a request when the controller is idle.  The ISR
; will complete the request (unless a verification error)
; and all other pending requests. When the ISR is finished
; all pending requests, it will signal the $DYDNE semaphore,
; unblocking this process.
;
;       R5 ->   Impure area (at request semaphore SDB)
;       R4      Modified
;
;-

        .enabl LSB

DYREQ:
        MOV     SAVSP(R5),SP            ; reset the stack in case of error
        CLR     ISRENB(R5)              ; isr is inactive
        waiq$s  sdb=R5,qelm=R4          ; Wait for a request packet
        BCC     2$                      ; Br if no error
1$:     CALL    $DDEXC                  ; report kernel error (code in R0)
                                        ; Does not return

        assum$  REQSEM EQ 0
2$:     MOV     R4,PACKET(R5)           ; Save -> packet in impure area
        CALL    DOREQ                   ; Verify & start request
        BCS     DYREQ                   ; br if verification error. Don't
                                        ; wait on the binary semaphore
                                        ; if the interrupt routine will
                                        ; not be active
        wait$s  sdb=$DYDNE(R5)          ; wait for controller to complete
                                        ; all requests
        BCS     1$                      ; br on errors

3$:     BR      DYREQ                   ; wait for another request
       .dsabl LSB

       .sbttl   DOREQ   - Verify and begin a request
```

```
;
; Verify and setup to start a request.  This routine has two
; returns:  A return with the carry clear indicates the request
; passed the verification tests and the ISR will become active.
; With the carry set on return, the request did not pass the
; verification tests and the ISR will not be started.
;
;   Inputs:
;           R5 -> impure area
;           R4 -> request packet
;
;           R3,R2,R1,R0 modified
;
;   CALL DOREQ
;
   .enabl LSB
DOREQ:
; Set up retry count, if not inhibited by function modifier
        MOVB   #DY$RTY,RTYCNT(R5)       ; Set up retry count.
        BIT    #FM$INH,DP.FUN(R4)       ; See if retry is inhibited.
        BEQ    10$                      ; If EQ, no, retry if errors
        CLRB    RTYCNT(R5)              ; Else inhibit retry on error
10$:
; Validate unit number   (ALL RX02'S HAVE MAX. OF TWO DRIVES)
        CLR     SIZE(R5)                ; clear flags
        MOV     #CSUN1,R3               ; Assume unit one is wanted.
        push$   R5                      ; Compute -> unit bit map
        ADD     #UNTMAP,(SP)            ;   "
        CLR     -(SP)                   ; Clear to convert byte to word
        MOVB    DP.UNI(R4),(SP)         ; Specified unit number
        BNE     20$                     ; Br if not unit zero
        CLR     R3                      ; It is unit zero

20$:    push$   #DY$MXU                 ; Maximum number of units
        CALL    $UNTVA                  ; Validate unit number
        TST     (SP)+                   ; Unit number valid?
        BNE     30$                     ; Br if so
        MOV     #ES$NXU,R2              ; Else non-existent unit error
        BR      90$                     ; Return message/process next request

30$:    CLR     ERROR(R5)               ; Clear error flags
        CLR     STATUS(R5)              ; and status flags
        MOV     R5,R2                   ; Compute -> density table
        ADD     #SAVDEN,R2              ;   " "
        CLR     -(SP)                   ; Change byte to word
        MOVB    DP.UNI(R4),(SP)         ; Get unit number
        ASL     (SP)                    ; get word offset
        ADD     (SP)+,R2                ; R2 -> density
        MOV     R2,DENPTR(R5)           ; save -> in impure area
        BIS     (R2),R3                 ; assume previous density
        MOV     #64.,R1                 ; sector size single density words
        TST     @DENPTR(R5)             ; Is it single density?
        BEQ     40$                     ; br if so
        ASL     R1                      ; make double density
40$:    MOV     R1,SECSIZ(R5)           ; save sector size
```

```
; Check function codes:
; 0,1=READ 3,4=WRITE 2,5=ILLEGAL 6=SET, 7=GET, 10+=ILLEGAL

        BIS     #CSGO!CSRD!CSINT,R3     ; assume read command
        MOV     DP.FUN(R4),R1           ; R1 = function code and modifiers
        BIC     #C77,R1         ; Clear all but function bits
        CMP     R1,#IF$RDL              ; write command?
        BLE     50$                     ; no
        ADD     #CSWRT-CSRD,R3          ; Make the read command a write
50$:    CMP     R1,#IF$GET              ; Check for legal range of code
        BHI     80$                     ; Br if illegal function
        MOVB    PLTABLE(R1),R0          ; Read or write function?
        BPL     100$                    ; Br if so
        ASLB    R0                      ; Get or set characteristics?
        BEQ     80$                     ; Br if not, a reserved code which
                                        ; we treat as illegal
        assum$  <IF$GET&1> EQ 1
        assum$  <IF$SET&1> EQ 0
        ASR     R1                      ; Get or set characteristics
        BCC     80$                     ; Br if set, which we don't support
        INCB    SIZE(R5)                ; flag get characteristics request
        BR      DYTRAN                  ; start request

60$:    MOV     #ES$IVP,R2              ; invalid parameter
        BR      90$
70$:    MOV     #ES$IDA,R2              ; invalid device address
        BR      90$
80$:    MOV     #ES$IFN,R2              ; Return illegal function code
90$:    MOV     R2,STATUS(R5)           ; set up error code
        CALL    REPLY                   ; signal user
        SEC                             ; special return
        RETURN                          ; Process next request

100$:   CMPB    R1,#IF$WTP              ; Physical write?
        BNE     120$                    ; Br if not
        BIT     #FM$WFM,DP.FUN(R4)      ; Format disk?
        BEQ     120$                    ; Br if not

; Format the diskette

        CMP     #"F0,DP.DAD(R4)         ; A safety check to verify that the
                                        ; user really wants to format the disk
        BNE     80$                     ; Br if he really didn't want to do it
                                        ; Return illegal function code
        BIC     #CSFUN,R3               ; clear function bits
        BIS     #CSFMT!CSINT!CSGO,R3    ; Set up format command
        BIS     #CSDN,R3                ; assume double density

        BIT     #FM$WSD,DP.FUN(R4)      ; Single density?

        BEQ     110$                    ; No
        BIC     #CSDN,R3                ; yes, clear density bit
110$:   INCB    FORMAT(R5)              ; set format flag
        BR      DYTRAN                  ; let's do it...


; Set up transfer...
;
; Compute extended address bits
```

```
120$:   MOV     DP.LEN(R4),R0           ; get number of bytes
        CLC                             ; set up for rotate
        ROR     R0                      ; convert to words
        BCS     60$                     ; br if odd byte count
        MOV     R0,WRDCNT(R5)           ; save it
        xtad$   vadd=DP.BUF(R4),par=DP.PAR(R4),pos=12.,ext=R2,addr=BUFFER(R5)

        BIS     R2,R3                   ; set xm bits
        MOVB    PLTABLE(R1),ADTYPE(R5)  ; See if this is physical or logical.
        BEQ     140$                    ; br if physical
        MOV     DP.DAD(R4),R2           ; get logical block number
        CMP     R2,#LBSIZE              ; too big?
        BGE     70$
        ASL     R2                      ; sector = block * 2 (double den)
        TST     @DENPTR(R5)             ; are we double density?
        BNE     130$                    ; br if yes
        ASL     R2                      ; sector = block * 4 single density
130$:   MOV     R2,LSN(R5)              ; save logical sector number
        BR      150$                    ; continue

140$:   MOV     DP.CYL(R4),R2
        CMP     R2,#NUMCYL              ; check argument
        BGE     70$
        MOV     R2,TRACK(R5)            ; physical access..get track
        MOV     DP.SEC(R4),R2
        CMP     R2,#NUMSEC              ; check sectors
        BGT     70$
        MOV     R2,SECTOR(R5)           ; save sector

150$:   .BR     DYTRAN                  ; start transfer

        .dsabl  LSB

        .sbttl  DYTRAN  - Start transfer or retry
;
;
;       Inputs:
;               R5 -> impure area
;               R3    funtion
;
;               R4,R2,R1,R0 modified
;
;        BR     DYTRAN
;
         .enabl LSB
DYTRAN:
        TSTB    FORMAT(R5)              ; format command?
        BEQ     1$                      ; nope
        MOV     #111,R2                 ; Magic number for format command
        CALL    INWAIT                  ; Do it
        BR      DYDONE                  ; all finished...
1$:
        TSTB    SIZE(R5)                ; Get characteristics?
        BEQ     30$                     ; no
; Copy device characteristics to the queue element
        MOV     #DTABLE,R0              ; -> Device characteristics table
        MOV     PACKET(R5),R1           ; -> Queue element
        ADD     #DP.DAD,R1              ; -> Device characteristics block
        MOV     #DTBSIZ,R2              ; = Number of bytes in table
```

```
20$:    MOVB    (R0)+,(R1)+             ; Copy data
        SOB     R2,20$                  ; Loop until done
        BIC     #CSFUN,R3               ; clear function bits
        BIS     #CSRDST!CSGO!CSINT,R3   ; set up read status
        CLR     R2                      ; no second command
        CALL    INWAIT                  ; wait for interrupt
        MOV     #LBSIZE,-(SP)           ; get # of blocks (dual density)
        BIT     #ESDDEN,@RX2DB(R5)      ; dual density?
        BNE     10$                     ; br if yes
        ASR     @SP                     ; single density

10$:    MOV     PACKET(R5),R1           ; R1 -> Packet
        MOV     (SP)+,DP.NLB(R1)        ; set size to correct value
        BIT     #ESDRDY,@RX2DB(R5)      ; drive ready?
        BNE     21$                     ; br if so
        MOV     #ES$UNS,STATUS(R5)      ; else drive unsafe
21$:    BR      DYDONE                  ; Return

30$:    BIT     #1*2,R3                 ; write function?
        BNE     40$                     ; no read
        CALL    DOSILO                  ; Fill silo for write
40$:    CALL    DOXFER                  ; do transfer to/from disk
        BIT     #1*2,R3                 ; read function?
        BEQ     50$                     ; No write
        CALL    DOSILO                  ; For read empty silo
50$:    MOV     COUNT(R5),R2            ; size of transfer (words)
        ASL     R2                      ; convert to bytes
        ADD     R2,BUFFER(R5)           ; update buffer address
        BCC     60$                     ; Br if carry clear
        ADD     #10000,R3               ; overflow into xm bits
;
; update values

60$:    TSTB    ADTYPE(R5)              ; physical accessing?
         BNE    70$                     ; br if logical
         INC    SECTOR(R5)              ; bump sector number
         CMP    #NUMSEC,SECTOR(R5)      ; Past end of track
         BGE    80$                     ; nope
         INC    TRACK(R5)               ; increment track number
         BR     80$

70$:     INC    LSN(R5)                  ; update logical sector number
80$:     SUB    COUNT(R5),WRDCNT(R5)     ; update word count
         BHI    30$                      ; br if more to do
         BIT    #1*2,R3                  ; read request?
         BNE    DYDONE                   ; yes, we are done
         TSTB   ADTYPE(R5)               ; Physical transfer?
         BEQ    DYDONE                   ; yes, no need to zero fill
;
; zero fill partial logical block

          MOV   #1,WRDCNT(R5)             ; set up 1 word transfer
          MOV   R5,R1                     ; Get -> to zero word
          ADD   #STATUS,R1                ;   "

; NOTE: xtad$ macro uses R0

       xtad$   vadd=R1,par=@#K.ISA3,pos=12.,ext=R4,addr=BUFFER(R5)
```

```
        BIC     #CSXM,R3                        ; clear xm bits
        BIS     R4,R3                           ; set new xm bits
        MOV     #3,R1                           ; test for block done
        BIT     #CSDN,R3                        ; double density?
        BEQ     90$                             ; no block # multiple of 4
        ASR     R1                              ; Block # even
90$:    BIT     R1,LSN(R5)                      ; are we through?
        BNE     30$                             ; nope...continue
       .BR      DYDONE

       .dsabl LSB

       .sbttl   DYDONE   - Finish processing a request

;
;               R5 -> impure area
;               R4    modified
;
;       BR      DYDONE
;
       .enabl   LSB
DYDONE:
       CLR      @RX2CSR(R5)             ; disable interrupts
       CALL     REPLY                   ; reply if needed
;
; test for new packets , start if any
; else signal semaphore and exit

10$:   waqc$s   sdb=R5,qelm=R4          ; another packet?
       BEQ      20$                     ; nope...go signal process
       MOV      R4,PACKET(R5)           ; yes, save -> new packet
       CALL     DOREQ                   ; start request
       BCS      10$                     ; illegal request...
                                        ; controller not started
       BR       30$                     ; otherwise return
20$:   sgnl$s   sdb=$DYDNE(R5)          ; Signal queue process
                                        ; controller is idle
30$:   RETURN                           ; Dismiss the real interrupt

       .dsabl LSB

       .sbttl   INWAIT   - Start function and wait for interrupt from floppy
;
;
;               R5 -> impure area
;               R3 command
;               R2 second command (0 if none)
;
;       CALL     INWAIT
;
;       Note:    Only registers R5 and R3 are preserved across
;                call to INWAIT
;
      .ENABL    LSB

INWAIT: MOV      (SP)+,INTRTN(R5)       ; save return point
        TST      @RX2CSR(R5)            ; is error bit set?   ****
        BMI      DYERR                  ; yes...check it out  ****
```

```
10$:    INC     ISRENB(R5)              ; flag main process is (will) blocked
        BEQ     10$                     ; if it rolled over..do it again
        MOV     R3,@RX2CSR(R5)          ; Start command
        MOV     R3,SAVR3(R5)            ; Save command for after intr
        TST     R2                      ; format or read status?
        BEQ     30$                     ; no..return
;
; Note: When running on an LSI, 4.5 micro seconds must elapse
; between loading the function and testing the transfer bit
;
20$:    BITB    #CSTRAN!CSDONE,@RX2CSR(R5) ; tr set?
        BEQ     20$                     ; no..try again
        MOV     R2,@RX2DB(R5)           ; save 2nd command
30$:
        CLC                             ; no special return
        RETURN                          ; does not return to caller
                                        ; but to caller's caller
        .DSABL  LSB

        .sbttl  DOSILO   - Initiate a silo fill or empty command

;
;       Inputs:
;
;               R5 -> impure area
;               R3    command
;
;       Outputs:
;               R4    word count this transfer
;               R2    buffer address
;
;       Note:   Only registers R5 and R3 are preserved across
;               call to DOSILO
;
        .ENABL  LSB

DOSILO: MOV     (SP)+,INTRTN(R5)        ; save return address
        MOV     WRDCNT(R5),R2           ; get word count for xfer
        BIC     #2*2,R3                 ; Change read/write to
                                        ; fill/empty command
        TST     @RX2CSR(R5)             ; is error bit set?   ****
        BMI     DYERR                   ; yes...check it out  ****
10$:    INC     ISRENB(R5)              ; flag main process is blocked
        BEQ     10$                     ; if it rolled over..do it again
        MOV     R3,@RX2CSR(R5)          ; start command

        BIS     #2*2,R3                 ; Fix command from above
        MOV     SECSIZ(R5),R4           ; get sector size
        CMP     R4,R2                   ; is sector size smaller
                                        ; than word count left?
        BLOS    20$                     ; yes, just read a sector
        MOV     R2,R4                   ; no, read remain count
20$:    MOV     R4,COUNT(R5)            ; save transfer count
        MOV     BUFFER(R5),R2           ; get  -> to buffer
        BR      DYDOFN                  ; go load count and
                                        ; address, start
        .DSABL  LSB
```

```
        .sbttl  DOXFER  - Start a sector read or write
;+
;       Inputs:
;               R3 command
;               R5 -> impure area
;
;       Outputs:
;               R4   sector
;               R2   track
;
;       Note:   Only registers R5 and R3 are preserved across
;               call to DOXFER
;-
        .ENABL  LSB

DOXFER: MOV     (SP)+,INTRTN(R5)        ; save return address
        TST     @RX2CSR(R5)             ; is error bit set?   ****
        BMI     DYERR                   ; yes...check it out  ****
10$:    INC     ISRENB(R5)              ; flag main process is blocked
        BEQ     10$                     ; if it rolled over..do it again
        MOV     R3,@RX2CSR(R5)          ; initiate function

; Note: When running on an LSI 4.5 micro seconds must elapse
; between loading the function and testing the transfer bit

        TSTB    ADTYPE(R5)              ; Physical transfer?
        BNE     20$                     ; No, we must compute interleave
        MOV     SECTOR(R5),R4           ; get sector in r4
        MOV     TRACK(R5),R2            ; track in r2
        BR      DYDOFN                  ; start operation

20$:    MOV     #8.,R2                  ; loop count
        MOV     LSN(R5),R4              ; Logical sector number
30$:    CMP     #26.*200,R4             ; Does 26 go into dividend?
        BHI     40$                     ; Br if not, c clear (bhi => bcc)
        ADD     #-26.*200,R4            ; Subtract 26 from dividend
        ;SEC                            ; C = 1 from 'add' above
40$:    ROL     R4                      ; Shift dividend and quotient
        DEC     R2                      ; dec loop count
        BGT     30$                     ; Br till divide done
        MOVB    R4,R2                   ; Copy track number
        CLRB    R4                      ; remove track number from remainder
        SWAB    R4                      ; get remainder
        CMP     #12.,R4                 ; c=1 if 13<=r4<=25, else c=0
        ROL     R4                      ; sector*2 (2:1 interleave)
                                        ; [+1 (c) if sector 13-25]
        ASL     R2                      ; double the track number
        ADD     R2,R4                   ; skew the sector
        ADD     R2,R4                   ; by adding in
        ADD     R2,R4                   ; 6 * track number
        ASR     R2                      ; undouble track number
        INC     R2                      ; and make it 1-76 (skip 0 for ansi)
50$:    SUB     #26.,R4                 ; Modulo sector into range 1-26
        BGE     50$                     ; loop until remainder goes neg
        ADD     #27.,R4                 ; put sector in range 1-26
      .BR       DYDOFN                  ; start transfer

      .DSABL    LSB


      .sbttl    DYDOFN  - Start a transfer or silo operation
```

```
;
;
;       Inputs:
;               R5 -> impure area
;               R4    first value (sector or word count)
;               R3    original function code
;               R2    second value (track or address)
;
        .ENABL  LSB

DYDOFN:
        MOV     R3,SAVR3(R5)            ; save command for after intr
;
; Note: When running on an LSI, 4.5 micro seconds must elapse
; between loading the function and testing the transfer bit
;
10$:    BITB    #CSTRAN,@RX2CSR(R5)     ; is TR set?
        BEQ     10$                     ; br if not
        MOV     R4,@RX2DB(R5)           ; load sector/word count
20$:    BITB    #CSTRAN,@RX2CSR(R5)     ; is TR set?
        BEQ     20$                     ; wait for TR
        MOV     R2,@RX2DB(R5)           ; now track/address
        CLC                             ; no special return
        RETURN                          ; bye

        .DSABL LSB

        .sbttl      $DYINT   - Interrupt Service Routine (ISR)
;+
;
; $DYINT - Interrupt Service Routine
;
; This code is invoked whenever an interrupt is received from the RX02.
; This routine restores R5 and R3 and jumps the routine which
; initiated the function.  If there are any errors, DYERR is called
; to investigate and possible retry.
; The ISR must be PIC code, so AMA is disabled here.
;
;-

$DYINT:
        fork$                           ; Request fork processing
        BCS     10$                     ; If CS, previous fork not processed
        MOV     R3,R5                   ; restore pointer to impure area
        MOV     SAVR3(R5),R3            ; restore r3 (command)
        TST     @RX2CSR(R5)             ; any errors?
        BMI     DYERR                   ; handle errors
        JMP     @INTRTN(R5)             ; continue where we left off
10$:    RETURN                          ; ???
```

```
        .sbttl      DYERR   - Error handler
;+
; Handles errors from RX02 and retrys function if retries are not
; inhibited.
;
;
;       R5 ->       Impure area
;       R3          command
;
;       BR          DYERR
;
;       R4,R2,R1,R0 modified.
;
;-

        .ENABL      LSB
DYERR:
        MOV         @RX2DB(R5),R0           ; save error register
        MOV         R0,COUNT(R5)            ; save it COUNT(R5)
        BIT         #ESDNER,R0              ; Density error?
        BEQ         20$                     ; no handle in usual manner
        MOV         DENPTR(R5),R2           ; -> density
        BIC         (R2)+,R3                ; Turn off bit if on
        NEGB        -(R2)                   ; tricky way to change 1 -> 0
        ;SEC/CLC                            ; c=1 if density was double, single
        INCB        (R2)+                   ; and 0 -> 1
        BIS         -(R2),R3                ; turn on double bit if not on
        BCS         10$                     ; br if it was double
        ASR         LSN(R5)                 ; single -> double 1/2 sector
        ASL         SECSIZ(R5)              ; double sector size
        BR          20$                     ; and continue

10$:    ASL         LSN(R5)                 ; double -> single 2* sector
        ASR         SECSIZ(R5)              ; 1/2 sector size

20$:    MOV         #CSINIT,@RX2CSR(R5)     ; start an initialize ...
                                            ; wait before setting ie since
                                            ; initialize clears csr


        DECB        RTYCNT(R5)              ; decrement retry count
        BLE         30$                     ; If LE, exhausted or inhibited

        BIT         #ESACLO,R0              ; was the error AC low?
        BNE         40$                     ; yes...any more would be useless
        MOV         R3,TEMP(R5)             ; save command
        MOV         #CSINT,R3               ; get interrupt enable
        CLR         R2                      ; no second command for inwait
        CALL        INWAIT                  ; do it
        MOV         TEMP(R5),R3             ; restore command
        MOV         COUNT(R5),R0            ; R0 = saved RX2DB
        BIT         #ESCRC!ESDNER,R0        ; Is it a CRC or density error ?
        BEQ         30$                     ; If EQ, no, it's a hard error
        JMP         DYTRAN                  ; Try again
```

```
;+
; Hard error, or retrys exhausted or inhibited.
; This code is to check drive ready bit which is
; valid only after a read status or initialize....
;-
30$:    MOV     #ESDRDY,R2              ; Get ready bit
        BIC     R2,R0                   ; clear drive not ready bit
        BIT     R2,@RX2DB(R5)           ; drive ready?
        BNE     40$                     ; Yes..retry if possible
        BIS     R2,R0                   ; set drive not read bit


40$:    push$   R0                      ; Copy error/status register

        MOV     #ERLIST,R0              ; -> list of status codes each error

        BIC     #170000!ESHEAD!ESUNIT!ESDDEN!ESDDAT!ESSID1,(SP)
                                        ; Clear bits which don't
                                        ; indicate errors
50$:    BEQ     60$                     ; Br if no further bits to check
        BIT     R1,(R0)+                ; Update error status pointer
        ASR     (SP)                    ; Set carry if error and Z if no more
                                        ; errors
        BCC     50$                     ; Br if not right error
60$:    TST     (SP)+                   ; Remove remnant of status register
        MOV     (R0),STATUS(R5)         ; Copy status indicator
        CMP     (R0),#ES$PWR            ; was AC low the error?
        BEQ     80$                     ; yes..any more is useless....

; This routine reads the extended error code from the RX02
; and places it in the error word to be shipped back to the user

        BIC     #CSXM!CSFUN,R3          ; clear function and xm bits
        BIS     #CSMAIN,R3              ; Read error reg
        MOV     R5,R2                   ; set up -> to buffer addr
        ADD     #SECTOR,R2              ;    "

; NOTE: xtad$ macro uses R0

        xtad$   vadd=R2,par=@#K.ISA3,pos=12.,ext=R4,addr=R1

        BIS     R4,R3                   ; Set new xm bits
        MOV     R1,R2                   ; set up low address
        CALL    INWAIT                  ; Go do it.
        MOVB    SECTOR(R5),ERROR(R5)    ; Put RX02's response in 'ERROR'.
70$:
        JMP     DYDONE

80$:    TST     ISRENB(R5)              ; is the main process blocked?
        BNE     70$                     ; br if so..unblock it
        CALL    REPLY                   ; otherwise reply
        JMP     DYREQ                   ; and get next request

       .dsabl   LSB
```

```
        .sbttl      REPLY   - Return Status Message Subroutine
;+
;
; This routine dispatches the reply to the user if he requested one .
;
;       R5      ->                  Impure area
;
;
;
;       CALL    REPLY
;
;-

        .enabl  LSB

        pure$

REPLY:
        MOV     PACKET(R5),R4               ; R4 -> request packet
        BEQ     40$                         ; no packet...exit

10$:    MOV     DP.LEN(R4),DP.ALN(R4)       ; Copy byte count requested
        MOV     STATUS(R5),DP.STS(R4)       ; Copy status code
        BEQ     20$                         ; If EQ, all bytes were xfered.
        assum$  ES$NOR EQ 0

        MOV     WRDCNT(R5),R2               ; # of words left in xfer
        ASL     R2                          ; make it bytes
        SUB     R2,DP.ALN(R4)               ; Subtract number left from total
20$:    MOV     ERROR(R5),DP.ERR(R4)        ; Put error code from READM in queue.
      CALL      $DRPLY                      ; reply to user
      CLR       PACKET(R5)                  ; no more packet
40$:  RETURN                                ; and return

     .dsabl     LSB

     .sbttl     DYSTP   - Request process termination routine

; This is the stop process section. It signals all the processes that are
; waiting that with the abort code and deletes the structures and the process.

DYSTP:
        CLR     @RX2CSR(R5)             ; Disable further interrupts
10$:    MOV     #ES$ABT,STATUS(R5)      ; Send abort status code to
        CALL    REPLY                   ; all waiting processes?

        waqc$s  sdb=R5,qelm=R4          ; any more requests?
        BCS   20$                       ; br if error
        BNE   10$                       ; yes...send abort code
20$:    dlst$s  sdb=R5                  ; Delete all structures
        dlst$s  sdb=$DYDNE(R5)          ; created.
        dint$s  vec=VECTOR(R5)          ; Disconnect from interrupt vector
        dlpc$


    .end
```

Index

A

AD driver
  features and capabilities, 7-1
  Get Characteristics function, 7-14
  prefix file, 7-15
  status codes, 7-15
ADPAR$ macro (Return PAR address), 15-3
Analog-to-digital conversions, 7-2
Ancillary Control Process (ACP)
  FALACP, 2-11
  features and capabilities, 2-1
  file I/O, 2-2
  prefix file, 2-9
  status codes, 2-8
Application development
  configuration guidelines, B-10
  initialization and self-test options, B-14
  KXT11-CA
    memory configuration steps, B-11
  partitioning, B-9
  peripheral processor, B-9
  tools
    MicroPower/Pascal, B-2
    summary, B-2
Application loading
  KXJ_LOAD, B-66
  KXT_LOAD, B-66
  peripheral processor, B-66
Applications
  overview
    peripheral processor, B-1
  peripheral processor, B-6
  software configuration, B-8
Arbiter process
  application, B-54
  communication with peripheral processor, B-8
  configuration file, B-54
  I/O page area, B-8
Arbiter processor
  device drivers, B-2
  LSI-11, B-1
  operating system environment, B-2
  peripheral processor relationship, B-1
Asynchronous I/O
  DDCMP, 12-3
  serial, 3-2
Automatic self-tests
  error reporting, B-18

B

$BLXIO subroutine (Block move), 15-25
Boot/Self-test switch, B-13, B-14
Bootstrap loader
  radial serial protocol (RSP), B-17
  TU58 DECtape II, B-17
Buffering, hardware, 3-20

C

Checksum
  calculating with DECprom, B-65
  specifying ROM test, B-16
Command register
  KW.DCO, B-22

Communication Device I/O
DECnet, 13-6
Communication driver
features and capabilities, 13-2
Communication support routines
peripheral processor, 13-32
Configuration file
CFDKTC.MAC, B-37
MicroPower/Pascal sample, B-37
CONFIGURATION macro, B-37
Configuration macro
device controller, 14-5
device driver, 14-4
Configuring Memory, B-10
Console ODT
hardware setup, B-18
Control and Status Register (CSR)
KX device driver, B-14
Conversions
analog-to-digital, 7-2
Copyright page
device driver, 14-11
Counter/Timer
support routines
external pulses, 6-21
KXT11–CA/KXJ11–CA, 6-9
linking counters, 6-24
CS driver
Disable Protocol function, 12-11
Enable Protocol function, 12-11
features and capabilities, 12-2
Get Characteristics function, 12-12
prefix file, 12-13
Read function, 12-11
status codes, 12-13
Write function, 12-11

D

Data transfers
DMA Transfer Controller (DTC), B-1
two-port RAM registers (TPR), B-1
DCT–11 microprocessor
general description, B-3
DD driver
Get Characteristics function, 4-22
DD driver (cont'd.)
Logical Read function, 4-20
Logical Write function, 4-20
Physical Read function, 4-21
Physical Write function, 4-21
$DDEXC subroutine (Report exception), 15-26
$DDINI macro (Device driver initialization), 15-27
Debugging
console ODT hardware setup, B-18
Declarations
device driver, 14-12
DECnet
communication device I/O, 13-6
DECprom
calculating ROM checksums, B-65
Device controller
configuration macro, CTRCF$, 14-5
errors
nonrecoverable, 14-19
recoverable, 14-19
Device controller process
device driver, 14-15
Device driver
arbiter processor, B-2
configuration macro, DRVCF$, 14-4
copyright page, 14-11
declarations, 14-12
device controller process, 14-15
error-processing routines, 14-18
errors
resource famine, 14-19
exception codes, 14-18
externally defined symbols, 14-12
functional description, 14-11
impure-area definition, 14-14
impure-area definition macro, xxISZ$, 14-9
initialization process, 14-14
invalid requests, 14-18
local macro definition, 14-12
macros
compute bus extended address, 15-22

Device driver
macros (cont'd.)
define driver packet symbols, 15-7
disable MMU context switch, 15-8
enable MMU context switch, 15-11
increment byte address, 15-13
increment word address, 15-14
move address and PAR, 15-18
move byte, 15-15
move byte (user-mode only), 15-16
move word, 15-19
move word (mapped case only), 15-17
move word (user-mode only), 15-20
read PAR or PDR register, 15-6
remap virtual address, 15-3
return PAR address, 15-3
set priority level, 15-21
write to PAR or PDR register, 15-10
module header, 14-11
overview, 14-1
peripheral processor, B-2
prefix module, 14-3
DYPFX.MAC, 14-8
priority assignments, 14-3
process definition, 14-12
pure-area definition, 14-14
reply subroutine, 14-17
sample MACRO–11 program, D-1
source module, 14-10
subroutines
allocate dynamic memory, 15-31
allocate memory, 15-28
block move, 15-25
deallocate dynamic memory, 15-29
initialize device driver, 15-27
initialize heap, 15-30
report exception, 15-26
Device driver
subroutines (cont'd.)
save/restore registers, 15-33
send device driver reply, 15-32
termination procedure, 14-18
Device I/O
Digital Network Architecture (DNA), 13-6
Device name, parsing, 2-10
DEVICES macro, B-37
Digital Network Architecture (DNA)
device I/O, 13-6
Disk drivers
features and capabilities, 4-2
prefix files, 4-26
status codes, 4-24
Disk I/O, 4-3
DL driver
Get Characteristics function, 4-12
Logical Read function, 4-10
Logical Write function, 4-10
Physical Read function, 4-11
Physical Write function, 4-11
DMA I/O
KXT11–CA/KXJ11–CA, 9-2
DMA Transfer Controller (DTC)
data transfers, B-1
DMA transfers
parallel I/O, 9-8, 9-21
sample program, 9-11
serial line unit, 9-8, 9-22
$DRALR subroutine (Allocate memory), 15-28
$DRDSP subroutine (Deallocate dynamic memory), 15-29
$DRHIN subroutine (Initialize heap), 15-30
DRMAP$ macro (Remap virtual address), 15-4
$DRNEW subroutine (Allocate dynamic memory), 15-31
DRPAR$ macro (Read PAR or PDR register), 15-6
$DRPLY subroutine (Send device driver reply), 15-32
DRVDF$ macro (Define driver packet symbols), 15-7

DSCXW$ macro (Disable MMU context switch), 15-8
DTC
  See DMA Transfer Controller (DTC)
DU driver
  Get Characteristics function, 4-18
  Logical Read function, 4-17
  Logical Write function, 4-17
DWPAR$ macro (Write to PAR or PDR register), 15-10
DY driver
  Get Characteristics function, 4-16
  Logical Read function, 4-13
  Logical Write function, 4-13
  Physical Read function, 4-14
  Physical Write function, 4-14
    format subfunctions, 4-15

E

ENCXW$ macro (Enable MMU context switch), 15-11
Error information, extended, 4-26, 6-44, 10-38
Error-processing routines
  device driver, 14-18
Errors
  automatic self-tests
    reporting, B-18
  device controller
    nonrecoverable, 14-19
    recoverable, 14-19
  device driver
    resource famine, 14-19
  KXT11-CA
    fatal, B-15
  self-tests
    reporting, B-15
Ethernet communication
  QN driver, 13-3
Exception codes
  device driver, 14-18
External pulses
  counter/timer support routines, 6-21

F

FALACP, 2-11
Features and capabilities
  AD driver, 7-1
  Ancillary Control Process (ACP), 2-1
  communication driver, 13-2
  CS driver, 12-2
  disk drivers, 4-2
  Instrument bus, 10-1
  KW driver, 8-1
  MU driver, 5-1
  Network Service Process (NSP), 11-1
  parallel line driver, 6-2
  QD driver, 9-1
  TT driver, 3-1
  XE driver, 10-3
File I/O
  Ancillary Control Process (ACP), 2-2
File system interface, Pascal, 2-3
File variable
  Get Characteristics request, 11-6
Fork routine
  Interrupt Service Routine (ISR), 14-17
Format subfunctions
  DY driver
    Physical Write function, 4-15
Functional description
  device driver, 14-11
Functions
  Allocate Channel
    QD driver, 9-24
  Auxiliary Command
    XE driver, 10-31
  Clear Timer
    YK driver, 6-43
  Close, 2-7
  Deallocate Channel
    QD driver, 9-24
  Delete, 2-7
  Disable
    KK driver, 13-24
    KX driver, 13-24
    XA driver, 6-31
    XP driver, 13-19
    XS driver, 13-19
  Disable Clock
    KW driver, 8-17
  Disable Portal

Functions
Disable Portal (cont'd.)
QN driver, 13-18
Disable Protocol
CS driver, 12-11
$DMA, 9-11
$DMA_ALLOCATE
QD driver, 9-11
$DMA_GET_STATUS
QD driver, 9-9
$DMA_SEARCH
QD driver, 9-6
$DMA_SEARCH_TRANSFER
QD driver, 9-7
$DMA_TRANSFER
QD driver, 9-4
DMA Complete
YK driver, 6-41
DMA Read
YK driver, 6-41
DMA Write
YK driver, 6-41
Enable
KK driver, 13-24
KX driver, 13-24
XA driver, 6-30
XP driver, 13-19
XS driver, 13-19
Enable Clock
KW driver, 8-15
Enable Portal
QN driver, 13-15
Enable Protocol
CS driver, 12-11
Enter, 2-6
Get Characteristics, 2-5
AD driver, 7-14
CS driver, 12-12
DD driver, 4-22
DL driver, 4-12
DU driver, 4-18
DY driver, 4-16
KK driver, 13-23
KW driver, 8-17
KX driver, 13-23
MU driver, 5-12
Network Service Process (NSP), 11-4
QD driver, 9-22
QN driver, 13-18
TT driver, 3-9
VM driver, 4-23

Functions
Get Characteristics (cont'd.)
XA driver, 6-30
XD driver, 4-19
XE driver, 10-25
XP driver, 13-20
XS driver, 13-20
YA driver, 6-32
YB driver, 6-36
YF driver, 6-37
YK driver, 6-39
Get Control
XE driver, 10-32
Go to Standby
XE driver, 10-33
KK_READ_DATA, 13-35
KK_WRITE_DATA, 13-35
KX_READ_DATA, 13-33
KX_WRITE_DATA, 13-34
Load Parallel Poll Register
XE driver, 10-29
Logical Read, 2-5
DD driver, 4-20
DL driver, 4-10
DU driver, 4-17
DY driver, 4-13
VM driver, 4-23
XD driver, 4-18
Logical Write, 2-5
DD driver, 4-20
DL driver, 4-10
DU driver, 4-17
DY driver, 4-13
VM driver, 4-23
XD driver, 4-18
Lookup, 2-6
Parallel Poll
XE driver, 10-29
Parallel Poll Configure
XE driver, 10-30
Pass Control
XE driver, 10-33
Physical Read, 2-4
DD driver, 4-21
DL driver, 4-11
DY driver, 4-14
Physical Write, 2-4
DD driver, 4-21
DL driver, 4-11
DY driver, 4-14
Protect, 2-7
Purge, 2-7

Functions (cont'd.)
Read
CS driver, 12-11
KK driver, 13-22
KX driver, 13-22
MU driver, 5-11
QD driver, 9-16
QN driver, 13-16
TT driver, 3-7
XA driver, 6-29
XP driver, 13-19
XS driver, 13-19
YA driver, 6-31
YB driver, 6-33
YF driver, 6-36
YK driver, 6-38
READ_PIO, 6-9
Read Logical
converted data, 7-13
XE driver, 10-23
Read Physical
KW driver, 8-13
Read Timer
YK driver, 6-43
Recognize Event
XE driver, 10-35
Rename, 2-7
Reposition Tape
MU driver, 5-12
Request Service
XE driver, 10-32
Rewind Tape
MU driver, 5-13
Serial Poll
XE driver, 10-28
SET_STATE
XE driver, 10-8
Set Characteristics, 2-5
configure device, 7-11
Network Service Process (NSP), 11-4
TT driver, 3-9
XE driver, 10-25
YB driver, 6-35
Set Event Mask
XE driver, 10-34
Set Modem Semaphore
TT driver, 3-14
XP driver, 13-21
XS driver, 13-21
Set Pattern
YK driver, 6-40
Functions (cont'd.)
Set Timer
YK driver, 6-42
Stop
XP driver, 13-21
XS driver, 13-21
Stop Request
TT driver, 3-15
Unprotect, 2-7
Wait for Event
XE driver, 10-35
Write
CS driver, 12-11
KK driver, 13-22
KX driver, 13-22
MU driver, 5-11
QD driver, 9-16
QN driver, 13-16
TT driver, 3-8
XA driver, 6-29
XE driver, 10-24
XP driver, 13-19
XS driver, 13-19
YA driver, 6-31
YB driver, 6-33
YF driver, 6-36
YK driver, 6-38
Write IEEE Remote Messages
XE driver, 10-27
Write Tape Mark
MU driver, 5-13
Write with EOI Termination
XE driver, 10-24
YK_CLEAR_TIMER, 6-21
YK_PORT_READ, 6-10
YK_PORT_WRITE, 6-11
YK_READ_TIMER, 6-20
YK_SET_PATTERN, 6-12
YK_SET_TIMER, 6-19

G

Get Characteristics function
file variable, 11-6
$SECTL Queue Semaphore, 11-5

H

Hardware
configuration
guidelines, B-10
peripheral processor, B-8

Hardware (cont'd.)
features
KXT11–CA, B-3
jumper
memory map, B-18
TPR base address, B-13
overview
peripheral processor, B-1
setup
peripheral processor, B-13
stand-alone processor, B-13
Hardware buffering, 3-20

I

I/O
asynchronous DDCMP, 12-3
asynchronous serial, 3-2
disk files, 4-3
DMA transfers, 9-2
instrument bus, 10-4
page area, B-8
parallel lines, 6-3
performing, 1-7
procedure interface, Pascal, 3-3, 4-5, 5-8, 6-5, 7-3, 13-10
real-time clock, 8-2
request/reply packets, 1-11
system architecture, 1-3
TMSCP tape files, 5-2
IBADR$ macro (Increment byte address), 15-13
Impure-area definition
device driver, 14-14
Impure-area definition macro
device driver, 14-9
Initialization options
selecting, B-14
Initialization process
device driver, 14-14
Instrument bus
features and capabilities, 10-1
I/O, 10-4
Interrupt Service Routine (ISR)
fork routine, 14-17
overview, 14-16
Invalid requests
device driver, 14-18
IWADR$ macro (Increment word address), 15-14

J

Jumper
memory map, B-18
TPR base address, B-13

K

KK_READ_DATA function, 13-35
KK_WRITE_DATA function, 13-35
KK driver
Disable function, 13-24
Enable function, 13-24
Get Characteristics function, 13-23
prefix file, 13-28
Read function, 13-22
status codes, 13-24
two-port RAM communication, 13-5
Write function, 13-22
KUI program, B-17
KW driver
Disable Clock function, 8-17
Enable Clock function, 8-15
features and capabilities, 8-1
Get Characteristics function, 8-17
prefix file, 8-18
Read Physical function, 8-13
status codes, 8-18
KX/KK protocol
command register definitions, B-27
concepts, B-22
driver transactions, B-23
interface initialization, B-32
KC.COM command field, B-27
KC.EOM bit, B-30
KC.IDA bit, B-28, B-29
KC.IDA command register bit, B-22
KC.IDR bit, B-28, B-30
KC.IDR command register bit, B-22
KC.LEN field, B-28, B-30
KC.NOP no-op command, B-27
KC.VEC field, B-28, B-30
KC$DI command, B-28
KC$EI command, B-28
KC$GS command, B-28
KC$RD command, B-28

KX/KK protocol (cont'd.)
KC$RSM command, B-27
KC$SS command, B-28
KC$WD command, B-29
KE$ILC code, B-31
KE$ILL code, B-31
KE$ILV code, B-31
KE$NDA code, B-31
KE$NDR code, B-31
KE$OK code, B-31
KE$OVR code, B-31
KS.ALN field, B-32
KS.DA bit, B-32
KS.DA status register bit, B-22
KS.DR bit, B-31
KS.EOM bit, B-31
KS.ERC field, B-31
KS.ERR bit, B-32
KS.IEN bit, B-28, B-32
KS.ON bit, B-32
KW.DCO register, B-22
master/slave relationship, B-19
message communication, B-25
overview, B-19
status register definitions, B-30
synchronizing operations, B-26
KX_READ_DATA function, 13-33
KX_WRITE_DATA function, 13-34
KX device driver
logical unit IDs, B-35
KX driver
Control and Status Register (CSR), B-14
Disable function, 13-24
Enable function, 13-24
Get Characteristics function, 13-23
prefix file, 13-28
Read function, 13-22
status codes, 13-24
two-port RAM communication, 13-5
Write function, 13-22
KXJ11-CA
See also Peripheral processor
application, B-54
configuration file, B-54
hardware
features, B-4
shared memory, B-52
stand-alone operation, B-1
KXJ_DISABLE_SHARED procedure, B-54
KXJ_ENABLE_SHARED procedure, B-53
KXJ_LOAD routine
application loading, B-66
loading KXJ11-CA, B-17
MIM File, B-66
user's interface, B-66
KXT11-CA
See also Peripheral processor
application loading, B-66
CSR assignments, B-33
DCT-11 microprocessor features, B-3
fatal error, B-15
hardware
features, B-3
interrupt vector assignments, B-33
loading from arbiter, B-17
memory
general description, B-3
memory configuration steps, B-11
stand-alone operation, B-1
KXT11-CA/KXJ11-CA
See Peripheral processor
KXT11C macro, B-37
KXT_LOAD routine
application loading, B-66
loading KXT11-CA, B-17
MIM file, B-66
program example, B-67
user's interface, B-66

L

LED display, B-18
fatal errors, B-15
Linking counters
counter/timer support routines, 6-24
Loading
KXJ11-CA
KXJ_LOAD routine, B-17
KXT11-CA
from arbiter, B-17
from RT-11 and RSX-11 systems, B-17
KXT_LOAD routine, B-17
TU58 DECtape II, B-17

Logical unit IDs
  KX device driver, B-35
Loopback tests, B-18
LSI–11 systems
  adding peripheral processors, B-7
  arbiter processor, B-1

M

Macro definition, local
  device driver, 14-12
Memory
  KXT11–CA
    configuration steps, B-11
    general description, B-3
    selecting maps, B-10
  map
    configuration rules, B-12
  map layout, B-11
MEMORY macro, B-37
Memory map
  jumper, B-18
    TPR base address, B-13
MicroPower/Pascal
  configuration guidelines, B-10
  device drivers, B-2
  features, B-2
  sample KXT11–CA configuration file, B-37
Module header
  device driver, 14-11
MU driver
  features and capabilities, 5-1
  Get Characteristics function, 5-12
  prefix file, 5-15
  Read function, 5-11
  Reposition Tape function, 5-12
  Rewind Tape function, 5-13
  status codes, 5-14
  Write function, 5-11
  Write Tape Mark function, 5-13
MVBYT$ macro, Move byte, 15-15
MVBYU$ macro, Move byte (user-mode only), 15-16
MVMAP$ macro, Move word (mapped case only), 15-17
MVVAD$ macro (Move address and PAR), 15-18
MVWRD$ macro, Move word, 15-19
MVWRU$ macro, Move word (user-mode only), 15-20

N

Network Service Process (NSP)
  features and capabilities, 11-1
  Get Characteristics function, 11-4
  prefix file, 11-8
  Set Characteristics function, 11-4
  status codes, 11-6
  task-to-task communication, 11-2
Node number, local
  determine and set, 11-15

P

Packet interface, request/reply
  overview, 1-8
Parallel I/O, 6-3
  DMA process
    KXT11–CA/KXJ11–CA, 6-15
  DMA transfers, 9-8, 9-21
  status codes, 6-43
  support routines
    KXT11–CA/KXJ11–CA, 6-9
    SBC–11/21, 6-8
Parallel line driver
  features and capabilities, 6-2
  prefix files, 6-45
Parallel processing, B-9
Pascal
  file system interface, 2-3, 11-4
  I/O procedure interface, 3-3, 4-5, 5-8, 6-5, 7-3, 12-6, 13-10
  support routines interface, 5-4, 6-7, 7-4, 8-3, 9-3, 10-5
Peripheral processor
  adding to LSI–11 systems, B-7
  application development, B-9
    design, B-9
    MicroPower/Pascal, B-2
    RT–11 and RSX tool kits, B-2
    tool kits, B-2
  applications, B-6
    overview, B-1
    partitioning, B-9

Peripheral processor
applications (cont'd.)
software configuration, B-8
arbiter processor relationship, B-1
communication support routines, 13-32
communication with arbiter process, B-8
configuring hardware, B-10
configuring software, B-10
device drivers, B-2
environment, B-1
configuring system, B-13
hardware
configuration, B-8
overview, B-1
setup, B-13
jumper
TPR base address, B-13
programming languages, B-2
Q-bus limits, B-13
software architecture, B-1
system ID switch, B-35
two-port RAM registers (TPR), B-1
XL driver, C-17
Prefix files
AD driver, 7-15
Ancillary Control Process (ACP), 2-9
CS driver, 12-13
disk drivers, 4-26
KK driver, 13-28
KW driver, 8-18
KX driver, 13-28
MU driver, 5-15
Network Service Process (NSP), 11-8
parallel line driver, 6-45
QD driver, 9-26
QN driver, 13-25
TT driver, 3-16
XA driver, 6-45
XE driver, 10-38
XL driver
KXT11-CA, C-28
PDP-11, C-12
XP driver, 13-26
XS driver, 13-26
YA driver, 6-46
YB driver, 6-47
Prefix files (cont'd.)
YF driver, 6-48
YK driver, 6-50
Prefix module
device driver, 14-3, 14-8
priority assignments, 14-3
PRIMITIVES macro, B-37
Priority assignments
device driver prefix module, 14-3
Procedures
IEQ_AUX_COMMAND
XE driver, 10-14
IEQ_COMMAND
XE driver, 10-10
IEQ_CONTROL_GTS
XE driver, 10-16
IEQ_PARALLEL_CONFIG
XE driver, 10-13
IEQ_PARALLEL_LOAD
XE driver, 10-13
IEQ_PARALLEL_POLL
XE driver, 10-12
IEQ_PASS_CONTROL
XE driver, 10-17
IEQ_REQ_SERVICE
XE driver, 10-15
IEQ_SERIAL
XE driver, 10-11
READ_ANALOG_SIGNAL, 7-7
READ_COUNTS_SIGNAL, 8-6
READ_COUNTS_WAIT, 8-3
READ_IEQ
XE driver, 10-6
READ_TAPE, 5-5
REC_IEQ_EVENT
XE driver, 10-18
REPOSITION_TAPE, 5-6
REWIND_TAPE, 5-7
SET_ANALOG_MODE, 7-5
SET_INT_MASK
XE driver, 10-17
SET_PIO_MODE, 6-8
START_RTCLOCK, 8-8
STOP_RTCLOCK, 8-10
WRITE_EOI_IEQ
XE driver, 10-9
WRITE_IEQ
XE driver, 10-7
WRITE_PIO, 6-8

Procedures (cont'd.)
WRITE_TAPE, 5-5
WRITE_TAPE_MARK, 5-7
Process definition
device driver, 14-12
PROCESSOR macro, B-37
Programming languages
peripheral processor, B-2
Pure-area definition
device driver, 14-14

Q

Q-bus
KXT11-CA limitations, B-13
QD driver
Allocate Channel function, 9-24
Deallocate Channel function, 9-24
$DMA_ALLOCATE function, 9-11
$DMA_DEALLOCATE function, 9-11
$DMA_GET_STATUS function, 9-9
$DMA_SEARCH_TRANSFER function, 9-7
$DMA_SEARCH function, 9-6
$DMA_TRANSFER function, 9-4
features and capabilities, 9-1
Get Characteristics function, 9-22
prefix file, 9-26
Read function, 9-16
status codes, 9-25
Write function, 9-16
QN driver
Disable Portal function, 13-18
Enable Portal function, 13-15
Ethernet communication, 13-3
Get Characteristics function, 13-18
prefix file, 13-25
Read function, 13-16
Write function, 13-16
Queue names, request, 1-9
Queue Semaphore
Get Characteristics, 11-5
Set Characteristics, 11-4

R

Radial serial protocol (RSP)
bootstrap loader, B-17
RAM
configuration rules, B-12
selecting maps, B-10
Random-access device
contiguous file storage, A-16
method, A-16
directory, A-4
entry, A-6
extended entry, A-8
fragmented, A-12
sample segment, A-9
segment header, A-5
end-of-segment marker, A-8
home block, A-2
size and number of files, A-18
structure, A-1
READ_ANALOG_SIGNAL procedure, 7-7
READ_COUNTS_SIGNAL procedure, 8-6
READ_COUNTS_WAIT procedure, 8-3
READ_PIO function, 6-9
READ_TAPE procedure, 5-5
Read Logical function
converted data, 7-13
Real-time clock I/O, 8-2
Reply subroutine
device driver, 14-17
REPOSITION_TAPE procedure, 5-6
Request/Reply packet interface, 2-4, 4-7, 5-9, 6-25, 7-8, 8-10, 9-14, 10-20, 12-8, 13-11
overview, 1-8
TT driver, 3-5
Request queue names, 1-9
RESOURCES macro, B-37
REWIND_TAPE procedure, 5-7
ROM
application start-up
selecting, B-16
calculating checksums, B-65
configuration rules, B-12
selecting maps, B-10
specifying checksum test, B-16

S

Sample program
  DMA transfers, 9-11
  task-to-task communication, 11-11, 11-13
SBC–11/21 PIO support routines, 6-8
$SECTL Queue Semaphore
  Get Characteristics, 11-5
  Set Characteristics, 11-4
Self-tests
  automatic, B-17
    error reporting, B-15
  ROM applications, B-16
  selecting options, B-14
Serial line unit
  DMA transfers, 9-8, 9-22
SET_ANALOG_MODE procedure, 7-5
SET_PIO_MODE procedure, 6-8
Set Characteristics function
  configure device, 7-11
  $SECTL Queue Semaphore, 11-4
SLU1
  loading programs from TU58 DECtape II, B-17
Software architecture
  master/slave concept, B-1
Source code
  XD driver, 4-30
Source module
  device driver, 14-10
SPL$ macro (Set priority level), 15-21
Stand-alone processor
  hardware setup, B-13
START_RTCLOCK procedure, 8-8
Starting
  ROM application, B-16
Status codes
  AD driver, 7-15
  Ancillary Control Process (ACP), 2-8
  CS driver, 12-13
  disk drivers, 4-24
  KK driver, 13-24
  KW driver, 8-18
  KX driver, 13-24
  MU driver, 5-14
  Network Service Process (NSP), 11-6
Status codes (cont'd.)
  parallel I/O, 6-43
  QD driver, 9-25
  TT driver, 3-15
  XE driver, 10-37
  XL driver, C-28
    PDP–11, C-12
STOP_RTCLOCK procedure, 8-10
$SV02 subroutine (Save/Restore registers), 15-33
$SV03 subroutine (Save/Restore registers), 15-33
$SV05 subroutine (Save/Restore registers), 15-33
Switch
  system ID, B-14
Symbols, externally defined
  device driver, 14-12
Synchronous serial I/O
  XP driver, 13-4
  XS driver, 13-4
System architecture, I/O, 1-3
System control registers
  two-port RAM registers (TPR), B-17
System ID switch, B-13
  selecting, B-14

T

Target system
  loading and starting, B-14
Task-to-task communication
  Network Service Process (NSP), 11-2
  sample program, 11-11, 11-13
Termination procedure
  device driver, 14-18
Tests
  automatic self-tests, B-17
  dedicated off-line, B-18
  loopback, B-18
  obtaining status information, B-17
TMSCP tape I/O, 5-2
TPR
  See Two-port RAM registers (TPR)
TRAPS macro, B-37
TT driver
  features and capabilities, 3-1
  Get Characteristics function, 3-9

TT driver (cont'd.)
prefix file, 3-16
Read function, 3-7
request/reply packet interface, 3-5
Set Characteristics function, 3-9
Set Modem Semaphore function, 3-14
status codes, 3-15
Stop Request function, 3-15
Write function, 3-8
TU58 DECtape II
bootstrap loader, B-17
Two-port RAM registers (TPR)
communication
KK driver, 13-5
KX driver, 13-5
disabling, B-13
enabling, B-13
peripheral processor, B-1
selecting base address, B-13
system control registers, B-17

V

VM driver
Get Characteristics function, 4-23
Logical Read function, 4-23
Logical Write function, 4-23

W

WRITE_PIO procedure, 6-8
WRITE_TAPE_MARK procedure, 5-7
WRITE_TAPE procedure, 5-5

X

XA driver
Disable function, 6-31
Enable function, 6-30
Get Characteristics function, 6-30
prefix file, 6-45
Read function, 6-29
Write function, 6-29
XD driver
Get Characteristics function, 4-19
Logical Read function, 4-18
Logical Write function, 4-18
source code, 4-30
XE driver
Auxiliary Command function, 10-31
features and capabilities, 10-3
Get Characteristics function, 10-25
Get Control function, 10-32
Go to Standby Function, 10-33
IEQ_AUX_COMMAND procedure, 10-14
IEQ_COMMAND procedure, 10-10
IEQ_CONTROL_GTS procedure, 10-16
IEQ_PARALLEL_CONFIG procedure, 10-13
IEQ_PARALLEL_LOAD procedure, 10-13
IEQ_PARALLEL_POLL procedure, 10-12
IEQ_PASS_CONTROL procedure, 10-17
IEQ_REQ_SERVICE procedure, 10-15
IEQ_SERIAL procedure, 10-11
Load Parallel Poll Register function, 10-29
Parallel Poll Configure function, 10-30
Parallel Poll function, 10-29
Pass Control function, 10-33
prefix file, 10-38
READ_IEQ procedure, 10-6
Read Logical function, 10-23
REC_IEQ_EVENT procedure, 10-18
Recognize Event function, 10-35
Request Service function, 10-32
Serial Poll function, 10-28
SET_INT_MASK procedure, 10-17
SET_STATE function, 10-8
Set Characteristics function, 10-25
Set Event Mask function, 10-34
status codes, 10-37
Wait for Event function, 10-35
WRITE_EOI_IEQ procedure, 10-9
WRITE_IEQ procedure, 10-7
Write function, 10-24

XE driver (cont'd.)
Write IEEE Remote Messages function, 10-27
Write with EOI Termination function, 10-24
XL Driver
Report Data-set Status Change function, C-28
XL driver
Block-Mode Read function, C-21
Block-Mode Write function, C-21
Connect Receive Ring Buffer function, C-21
Connect Transmit Ring Buffer function, C-21
function-dependent request formats, C-5, C-20
Get Status function, C-25
KXT11-CA
prefix file, C-28
PDP-11, C-1
Connect Receive Ring Buffer function, C-4
Connect Transmit Ring Buffer function, C-4
device-independent function modifiers, C-5
Disconnect Receive Ring Buffer function, C-4
Disconnect Transmit Ring Buffer function, C-4
functions, C-3
Get Status function, C-5
prefix file, C-12
Read function, C-3
Report Data-Set Status Change function, C-4
Set Status Function, C-5
status codes, C-12
Write Function, C-3
peripheral processor, C-17
Connect Receive Ring Buffer function, C-19
Connect Transmit Ring Buffer function, C-19
device-independent function modifiers, C-20
XL driver
peripheral processor (cont'd.)
Disconnect Receive Ring Buffer function, C-19
Disconnect Transmit Ring Buffer function, C-20
functions provided, C-18
Get Status function, C-20
Read function, C-19
Report Data-Set Status Change function, C-20
Set Status function, C-20
Write function, C-19
Ring Buffer Disconnect function, C-22
Set Status function, C-22
status codes, C-28
XP driver
Disable function, 13-19
Enable function, 13-19
Get Characteristics function, 13-20
prefix file, 13-26
Read function, 13-19
Set Modem Semaphore function, 13-21
Stop function, 13-21
synchronous serial I/O, 13-4
Write function, 13-19
XS driver
Disable function, 13-19
Enable function, 13-19
Get Characteristics function, 13-20
prefix file, 13-26
Read function, 13-19
Set Modem Semaphore function, 13-21
Stop function, 13-21
synchronous serial I/O, 13-4
Write function, 13-19
XTAD$ macro (Compute Bus Extended Address), 15-22

Y

YA driver
Get Characteristics function, 6-32
prefix file, 6-46
Read function, 6-31

YA driver (cont'd.)
  Write function, 6-31
YB driver
  Get Characteristics function, 6-36
  prefix file, 6-47
  Read function, 6-33
  Set Characteristics function, 6-35
  Write function, 6-33
YF driver
  Get Characteristics function, 6-37
  prefix file, 6-48
  Read function, 6-36
  Write function, 6-36
YK_CLEAR_TIMER function, 6-21
YK_PORT_READ function, 6-10
YK_PORT_WRITE function, 6-11
YK_READ_TIMER function, 6-20
YK_SET_PATTERN function, 6-12
YK_SET_TIMER function, 6-19
YK driver
  Clear Timer function, 6-43
  DMA Complete function, 6-41
  DMA Read function, 6-41
  DMA Write function, 6-41
  Get Characteristics function, 6-39
  prefix file, 6-50
  Read function, 6-38
  Read Timer function, 6-43
  Set Pattern function, 6-40
  Set Timer function, 6-42
  Write function, 6-38

HOW TO ORDER
ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008
Rest of U.S.A. and Puerto Rico*	800–258–1710	Nashua, NH 03061
* Prepaid orders from Puerto Rico must be placed with DIGITAL's local subsidiary (809–754–7575)		
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	617–234–4323	Publishing & Circulation Serv. (P&CS) NR03–1/W3 Digital Equipment Corporation Northboro, MA 01532

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

— Do Not Tear — Fold Here and Tape —

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— Do Not Tear — Fold Here —

Cut Along Dotted Line