

**UPDATE NOTICE 1**

**MicroPower/Pascal™**

**Runtime Services Manual**

**AA-M391B-T1**

**February 1984**

**NEW AND CHANGED INFORMATION**

This update contains changes and additions to the *MicroPower Pascal Runtime Services Manual*, AA-M391B-TC.

To order additional documents from within DIGITAL, contact the Software Distribution Center, Northboro, Massachusetts 01532

To order additional documents from outside DIGITAL, refer to the instructions at the back of this document.

**digital equipment corporation · maynard, massachusetts**

## INSTRUCTIONS

The enclosed pages are replacements for or additions to current pages of the *MicroPower/Pascal Runtime Services Manual*. On replacement pages, changes and additions are indicated by vertical bars (|); deletions are indicated by bullets (●).

Keep this notice in your manual to maintain an up-to-date record of changes.

© Digital Equipment Corporation 1984.

All Rights Reserved.

Printed in U.S.A.

Old page	New page
Title/Copyright	Title/Copyright
iii-xi	iii-xii
xiii/xiv	xiii xiv
xv/blank	xv/blank
1-11/1-12	1-11 1-12
3-9/3-10 to 3-13 3-14	3-9 3-10 to 3-13 3-14
3-25/3-26	3-25 3-26
3-27/3-28	3-27 3-28
	3-28.1/blank
3-43/3-44	3-43/3-44
3-53/3-54	3-53 3-54
3-61/3-62	3-61/3-62
3-65/3-66	3-65/3-66
	3-66.1/blank
3-71/3-72	3-71/3-72
3-73/3-74	3-73 3-74
3-85/3-86	3-85/3-86
3-91/3-92	3-91/3-92
	3-92.1/blank
3-93/3-94 to 3-97 3-98	3-93/3-94 to 3-97 3-98
3-101/3-102	3-101/3-102
3-103/3-104	3-103/3-104
	3-104.1/blank
3-107/3-108	3-107/3-108
	3-109/blank
4-1/4-2 to 4-83 blank	4-1/4-2 to 4-127 4-128
8-1/8-2 to 8-5 8-6	8-1/8-2 to 8-5 8-6
9-5/9-6	9-5/9-6
9-7/9-8	9-7/9-8
	9-8.1/9-8.2
9-9/9-10	9-9/9-10
9-15/9-16	9-15/9-16
G-3/G-4	G-3 G-4
G-13/G-14	G-13/G-14
	G-14.1/blank
G-15/G-16	G-15/G-16
G-17/G-18	G-17/G-18
	H-1/H-2 to H-24
Index-1/Index-2 to Index-9 blank	Index-1/Index-2 to Index-9/Index-10
Reader's Comments Mailer	Reader's Comments Mailer

## CONTENTS

		Page
PREFACE		xiii
CHAPTER 1	INTRODUCTION	1-1
1.1	THE MICROPOWER/PASCAL RUNTIME SYSTEM	1-1
1.2	KERNEL ORGANIZATION	1-2
1.2.1	Overview of Primitive Services	1-2
1.2.1.1	Process-Management Primitives	1-3
1.2.1.2	Resource-Management Primitives	1-5
1.2.1.3	Process-Synchronization Primitives	1-5
1.2.1.4	Message-Transmission Plus Synchronization Primitives	1-7
1.2.1.5	Ring Buffer Primitives	1-8
1.2.1.6	Exception-Processing Primitives	1-10
1.2.1.7	Interrupt-Management Primitives	1-10
1.2.2	Overview of System Processes	1-11
1.2.2.1	Standard Device Handlers	1-11
1.2.2.2	Clock Process	1-12
CHAPTER 2	PROCESSES AND SYSTEM DATA STRUCTURES	2-1
2.1	PROCESSES	2-1
2.1.1	Static and Dynamic Processes	2-2
2.1.2	Process Names	2-4
2.1.3	Process States	2-5
2.1.3.1	Process State Codes	2-7
2.1.3.2	State Queues	2-8
2.1.4	Process Scheduling	2-8
2.1.4.1	Process Preemption	2-8
2.1.4.2	Process Blocking and Unblocking	2-9
2.1.4.3	Process Suspension	2-10
2.1.4.4	Exception Handling	2-11
2.1.4.5	Scheduler	2-12
2.1.5	Process Control Block (PCB)	2-12
2.1.6	Memory Partitioning and Process/Program Segmentation	2-16
2.1.7	Process Mapping Types	2-18
2.2	SYSTEM DATA STRUCTURES	2-23
2.2.1	Typed Data Structures	2-24
2.2.1.1	Structure Names and Name Blocks	2-24
2.2.1.2	Structure Header	2-25
2.2.1.3	Binary Semaphore Definition	2-26
2.2.1.4	Counting Semaphore Definition	2-27
2.2.1.5	Queue Semaphore Definition	2-27
2.2.1.6	Ring Buffer Definition	2-28
2.2.1.7	Process Control Block Definition	2-30
2.2.1.8	Unformatted Structure Definition	2-30
2.2.2	Message Packets	2-30

2.2.3	System Queues	2-31	
2.2.3.1	Singly Linked Lists	2-32	
2.2.3.2	Doubly Linked Lists	2-33	
2.2.4	System-Common Memory Organization	2-33	
CHAPTER	3	MACRO-11 PRIMITIVE SERVICE REQUESTS	3-i
3.1	GENERAL CONVENTIONS AND USAGE RULES	3-1	
3.1.1	Macro Variant prim\$	3-2	
3.1.2	Macro Variant prim\$S	3-3	
3.1.3	Macro Variant prim\$P	3-4	
3.1.4	Error Returns	3-5	
3.1.5	Structure Descriptor Block (SDB) Usage	3-5	
3.1.5.1	Initialization of SDBs for Named Structures	3-7	
3.1.5.2	Initialization of SDBs for Unnamed Structures	3-7	
3.1.6	Process Descriptor Block (PDB) Usage	3-7	
3.2	ALPC\$ (CONDITIONALLY ALLOCATE PACKET)	3-10	
3.3	ALPK\$ (ALLOCATE PACKET)	3-12	
3.4	CCND\$ (CONNECT TO EXCEPTION CONDITION)	3-14	
3.5	CHGPs\$ (CHANGE PROCESS PRIORITY)	3-17	
3.6	CINT\$ (CONNECT TO INTERRUPT)	3-19	
3.7	CRPC\$ (CREATE PROCESS)	3-22	
3.8	CRST\$ (CREATE STRUCTURE)	3-26	
3.9	DAPK\$ (DEALLOCATE PACKET)	3-29	
3.10	DEXC\$ (DISMISS EXCEPTION CONDITION)	3-31	
3.11	DFSPCS\$ (DEFINE STATIC PROCESS)	3-33	
3.12	DINT\$ (DISCONNECT FROM INTERRUPT)	3-35	
3.13	DLPC\$ (DELETE PROCESS)	3-39	
3.14	DLST\$ (DELETE STRUCTURE)	3-43	
3.15	FORK\$ (FORK PROCESSING)	3-42	
3.16	GELCS (CONDITIONAL GET ELEMENT)	3-44	
3.17	GELMS (GET ELEMENT)	3-46	
3.18	GTSTS (GET PROCESS STATUS)	3-48	
3.19	GVAL\$ (RETURN STRUCTURE VALUE)	3-50	
3.20	IMPUR\$ (DEFINE AN IMPURE PROGRAM DATA SECTION)	3-52	
3.21	PDAT\$ (DEFINE A PURE PROGRAM DATA SECTION)	3-53	
3.22	PELCS (CONDITIONAL PUT ELEMENT)	3-54	
3.23	PELM\$ (PUT ELEMENT)	3-56	
3.24	PURE\$ (DEFINE A PURE PROGRAM INSTRUCTION SECTION)	3-58	
3.25	P7SYSS\$ (DROP CPU PRIORITY)	3-59	
3.26	RBUFS\$ (RESET RING BUFFER)	3-60	
3.27	RCVC\$ (CONDITIONAL RECEIVE DATA)	3-62	
3.28	RCVD\$ (RECEIVE DATA)	3-66.1	
3.29	REXC\$ (REPORT EXCEPTION)	3-70	
3.30	RSUM\$ (RESUME PROCESS)	3-72	
3.31	SALL\$ (SIGNAL ALL WAITERS)	3-74	
3.32	SCHDS (SCHEDULE PROCESS)	3-76	
3.33	SEND\$ (SEND DATA)	3-77	
3.34	SERA\$ (SET EXCEPTION ROUTINE ADDRESS)	3-82	
3.35	SGLCS (CONDITIONALLY SIGNAL SEMAPHORE)	3-85	
3.36	SGLQS\$ (SIGNAL QUEUE SEMAPHORE)	3-87	
3.37	SGNLS\$ (SIGNAL SEMAPHORE)	3-89	
3.38	SGQCS\$ (CONDITIONALLY SIGNAL QUEUE SEMAPHORE)	3-91	
3.39	SNDC\$ (CONDITIONAL SEND DATA)	3-93	
3.40	SPND\$ (SUSPEND PROCESS)	3-97	
3.41	STPC\$ (STOP PROCESS)	3-99	
3.42	WAICS (CONDITIONALLY WAIT ON SEMAPHORE)	3-101	
3.43	WAIQS\$ (WAIT ON QUEUE SEMAPHORE)	3-103	
3.44	WAIT\$ (WAIT ON SEMAPHORE)	3-105	
3.45	WAQCS\$ (CONDITIONAL WAIT ON QUEUE SEMAPHORE)	3-107	

CHAPTER	4	STANDARD DEVICE HANDLERS	4-1
4.1		GENERAL DEVICE-HANDLER INTERFACE	4-2
4.1.1		Request Queue Names	4-3
4.1.2		I/O Request Packet	4-5
4.1.2.1		Sample I/O Request in Pascal	4-9
4.1.2.2		Sending an I/O Request in MACRO-11	4-11
4.1.3		Completion Reply Format	4-12
4.1.3.1		Receiving an I/O Reply Message in Pascal	4-15
4.1.3.2		Receiving an I/O Reply Message in MACRO-11	4-16
4.2		ADV -C/AXV11-C (AA) ANALOG INPUT CONVERTER HANDLER	4-17
4.2.1		Functions Provided	4-18
4.2.1.1		Get Characteristics Function	4-18
4.2.1.2		Connect Receive Ring Buffer Function	4-18
4.2.1.3		Stop Function	4-18
4.2.1.4		Read Converted Data Function	4-19
4.2.1.5		Continuous Read Function	4-19
4.2.1.6		Device-Dependent Function Modifiers	4-19
4.2.1.7		Device-Independent Function Modifiers	4-19
4.2.2		Function-Dependent Request Formats	4-19
4.2.2.1		Get Characteristics Function	4-19
4.2.2.2		Connect Receive Ring Buffer Function	4-20
4.2.2.3		Stop Function	4-20
4.2.2.4		Read Converted Data Function	4-20
4.2.2.5		Continuous Read Function	4-23
4.2.3		Status Codes	4-25
4.2.4		Extended Error Information	4-25
4.3		TU58 (DD) DEVICE HANDLER	4-26
4.3.1		Functions Provided	4-27
4.3.1.1		Device-Dependent Function Modifiers	4-28
4.3.1.2		Device-Independent Function Modifiers	4-28
4.3.2		Function-Dependent Request Formats	4-28
4.3.2.1		Logical Read/Write Functions	4-29
4.3.2.2		Physical Read/Write Functions	4-29
4.3.2.3		Get Characteristics Function	4-30
4.3.3		Status Codes	4-30
4.3.4		Extended Error Information	4-30
4.4		RL01/RL02 (DL) DEVICE HANDLER	4-31
4.4.1		Functions Provided	4-32
4.4.1.1		Device-Dependent Function Modifiers	4-34
4.4.1.2		Device-Independent Function Modifiers	4-34
4.4.2		Function-Dependent Request Formats	4-34
4.4.2.1		Logical Read/Write Functions	4-34
4.4.2.2		Physical Read/Write Functions	4-35
4.4.2.3		Get Characteristics Function	4-35
4.4.3		Status Codes	4-36
4.4.4		Extended Error Information	4-36
4.5		MSCP DISK-CLASS (DU) DEVICE HANDLER	4-37
4.5.1		Functions Provided	4-38
4.5.1.1		Device-Dependent Function Modifiers	4-39
4.5.1.2		Device-Independent Function Modifiers	4-39
4.5.2		Function-Dependent Request Formats	4-39
4.5.2.1		Logical Read/Write Functions	4-39
4.5.2.2		Get Characteristics Function	4-40
4.5.2.3		Bypass Only Function	4-40
4.5.2.4		Bypass Function	4-40
4.5.2.5		Initialize Port Function	4-41
4.5.3		Status Codes	4-41
4.6		RX02 (DY) DEVICE HANDLER	4-43
4.6.1		Functions Provided	4-44
4.6.1.1		Device-Dependent Function Modifiers	4-45
4.6.1.2		Device-Independent Function Modifiers	4-45
4.6.2		Function-Dependent Request Formats	4-45

4.6.2.1	Logical Read/Write Functions	4-46
4.6.2.2	Physical Read/Write Functions	4-46
4.6.2.3	Format Subfunctions of Physical Write	4-47
4.6.2.4	Get Characteristics Function	4-47
4.6.3	Status Codes	4-48
4.6.4	Extended Error Information	4-48
4.7	KWV11-C (KW) REAL-TIME CLOCK HANDLER	4-49
4.7.1	Functions Provided	4-49
4.7.1.1	Device-Dependent Function Modifiers	4-50
4.7.1.2	Device-Independent Function Modifiers	4-50
4.7.2	Function-Dependent Request Formats	4-53
4.7.2.1	Read Physical Function	4-52
4.7.2.2	Get Characteristics Function	4-53
4.7.2.3	Start Real-Time Clock Function	4-53
4.7.2.4	Stop Real-Time Clock Function	4-53
4.7.3	Status Codes	4-54
4.7.4	Extended Error Information	4-54
4.8	KXT11-C TWO-PORT RAM (KX) HANDLER	4-55
4.8.1	Functions Provided	4-5
4.8.1.1	Device-Independent Function Modifiers	4-57
4.8.2	Function-Dependent Request Formats	4-57
4.8.2.1	Read or Write Functions	4-58
4.8.2.2	Get Characteristics Function	4-58
4.8.3	Status Codes	4-58
4.9	DRV11-J PARALLEL-LINE (XA) DEVICE HANDLER	4-59
4.9.1	Functions Provided	4-63
4.9.1.1	Device-Dependent Function Modifiers	4-64
4.9.1.2	Device-Independent Function Modifiers	4-61
4.9.2	Function-Dependent Request Formats	4-61
4.9.2.1	Read or Write Functions	4-61
4.9.2.2	Get Characteristics Function	4-61
4.9.2.3	Enable Function	4-62
4.9.2.4	Disable Function	4-62
4.9.3	Status Codes	4-62
4.10	SERIAL LINE (XL) DEVICE HANDLER	4-63
4.10.1	Functions Provided	4-64
4.10.1.1	Read Function	4-65
4.10.1.2	Write Function	4-65
4.10.1.3	Connect Receive Ring Buffer Function	4-65
4.10.1.4	Disconnect Receive Ring Buffer Function	4-66
4.10.1.5	Connect Transmit Ring Buffer Function	4-66
4.10.1.6	Disconnect Transmit Ring Buffer Function	4-66
4.10.1.7	Report Data-Set Status Change Function	4-66
4.10.1.8	Set Status Function	4-67
4.10.1.9	Get Status Function	4-67
4.10.1.10	Device-Independent Function Modifiers	4-67
4.10.2	Function-Dependent Request Formats	4-67
4.10.2.1	Block-Mode Read or Write Functions	4-68
4.10.2.2	Connect Receive or Transmit Ring Buffer Functions	4-68
4.10.2.3	Disconnect Receive or Transmit Ring Buffer Functions	4-69
4.10.2.4	Set Status Function	4-69
4.10.2.5	Get Status Function	4-71
4.10.2.6	Report Data-Set Status Change Function	4-72
4.10.3	Status Codes	4-73
4.11	DPV11 COMMUNICATIONS LINE (XP) DEVICE HANDLER	4-74
4.11.1	Functions Provided	4-74
4.11.2	Function-Dependent Request Formats	4-76
4.11.2.1	Block-Mode Read or Write Functions	4-76
4.11.2.2	Connect Link Function	4-77
4.11.2.3	Disconnect Link Function	4-77
4.11.2.4	Set Characteristics Function	4-77

4.11.2.5	Get Characteristics Function	4-77
4.11.2.6	Set Modem Status	4-79
4.11.2.7	Sense Modem Status	4-80
4.11.3	Status Codes	4-81
4.11.4	X.25 Protocol Message Frame Format	4-81
4.12	DRV11 PARALLEL-LINE (YA) DEVICE HANDLER	4-84
4.12.1	Functions Provided	4-87
4.12.1.1	Get Characteristics Function	4-87
4.12.1.2	Device-Dependent Function Modifiers	4-88
4.12.1.3	Device-Independent Function Modifiers	4-88
4.12.2	Function-Dependent Request Formats	4-88
4.12.2.1	Read or Write Functions	4-87
4.12.2.2	Get Characteristics Function	4-87
4.12.3	Status Codes	4-87
4.13	SBC-11/21 PARALLEL PORT (YF) HANDLER	4-88
4.13.1	Functions Provided	4-89
4.13.1.1	Device-Dependent Function Modifiers	4-89
4.13.1.2	Device-Independent Function Modifiers	4-90
4.13.2	Function-Dependent Request Formats	4-90
4.13.2.1	Read or Write Functions	4-90
4.13.2.2	Get Characteristics Function	4-90
4.13.3	Status Codes	4-91
4.14	KXT11-C TU58 (DD) DEVICE HANDLER	4-92
4.15	KXT11-C TWO-PORT RAM (KK) HANDLER	4-93
4.15.1	Functions Provided	4-94
4.15.1.1	Device-Independent Function Modifiers	4-94
4.15.2	Function-Dependent Request Formats	4-94
4.15.2.1	Read or Write Functions	4-95
4.15.2.2	Get Characteristics Function	4-95
4.15.3	Status Codes	4-95
4.16	KXT11-C DMA TRANSFER CONTROLLER (QD) HANDLER	4-96
4.16.1	Functions Provided	4-96
4.16.1.1	Device-Dependent Function Modifiers	4-97
4.16.1.2	Device-Independent Function Modifiers	4-97
4.16.2	Function-Dependent Request Formats	4-97
4.16.2.1	Read and Write Functions	4-97
4.16.2.2	Get Characteristics Function	4-98
4.16.2.3	Allocate Channel Function	4-102
4.16.2.4	Deallocate Channel Function	4-103
4.16.3	Status Codes	4-103
4.17	KXT11-C ASYNCHRONOUS SERIAL LINE (XL) HANDLER	4-105
4.17.1	Functions Provided	4-106
4.17.1.1	Read Function	4-106
4.17.1.2	Write Function	4-106
4.17.1.3	Connect Receive Ring Buffer Function	4-107
4.17.1.4	Disconnect Receive Ring Buffer Function	4-107
4.17.1.5	Connect Transmit Ring Buffer Function	4-107
4.17.1.6	Disconnect Transmit Ring Buffer Function	4-107
4.17.1.7	Set Status Function	4-107
4.17.1.8	Get Status Function	4-108
4.17.1.9	Report Data-Set Status Change Function	4-108
4.17.1.10	Device-Independent Function Modifiers	4-108
4.17.2	Function-Dependent Request Formats	4-108
4.17.2.1	Block-Mode Read or Write Functions	4-109
4.17.2.2	Connect Receive or Transmit Ring Buffer Functions	4-109
4.17.2.3	Disconnect Receive or Transmit Ring Buffer Functions	4-109
4.17.2.4	Set Status Function	4-110
4.17.2.5	Get Status Function	4-112
4.17.2.6	Report Data-Set Status Change Function	4-114
4.17.3	Status Codes	4-115
4.18	KXT11-C SYNCHRONOUS SERIAL LINE (XS) HANDLER	4-116

4.18.1	Functions Provided	4-117	
4.18.2	Function-Dependent Request Formats	4-118	
4.18.2.1	Get Frame and Send Frame Functions	4-118	
4.18.2.2	Kill Requests Function	4-118	
4.18.3	Status Codes	4-119	
4.19	KXT11-C PARALLEL PORT AND TIMER/COUNTER (YK)		
	HANDLER	4-120	
4.19.1	Functions Provided	4-123	
4.19.1.1	Device-Dependent Function Modifiers	4-124	
4.19.2	Function-Dependent Formats	4-124	
4.19.2.1	Read Functions	4-125	
4.19.2.2	Write Functions	4-125	
4.19.2.3	Get Characteristics Function	4-125	
4.19.2.4	Pattern Set Function	4-126	
4.19.2.5	Timer Set Function	4-127	
4.19.2.6	Timer Read Function	4-127	
4.19.3	Status Codes	4-128	
<b>CHAPTER</b>	<b>5</b>	<b>MICROPOWER/PASCAL FILE SYSTEM</b>	<b>5-1</b>
5.1	MACRO-11 FILE INPUT AND OUTPUT	5-1	
5.2	SQIO SUBROUTINE	5-1	
5.2.1	SQIO Read and Write Functions	5-2	
5.2.1.1	Virtual Reads and Writes	5-2	
5.2.1.2	Logical Reads and Writes	5-3	
5.2.1.3	Physical Reads and Writes	5-3	
5.2.1.4	Function Modifiers	5-3	
5.2.2	SQIO Directory Operations	5-4	
5.2.2.1	File-Opening Functions	5-4	
5.2.2.2	File-Closing Functions	5-5	
5.2.2.3	File Utility Functions	5-5	
5.2.3	Device Driver Process Replies	5-6	
5.2.4	File Status	5-6	
5.3	PARSING FILE SPECIFICATIONS	5-6	
5.3.1	File Specification Syntax	5-7	
5.3.2	File Specification Examples	5-7	
5.4	PREDEFINED MACROS	5-8	
5.4.1	FSINT\$	5-8	
5.4.2	QIOS	5-8	
5.4.2.1	Macro Syntax	5-8	
5.4.2.2	Syntax Examples	5-9	
5.4.3	FIB\$	5-9	
5.4.3.1	File Identification Block Fields	5-10	
5.4.3.2	Macro Syntax	5-14	
5.4.4	PARSE\$	5-14	
5.4.4.1	Macro Syntax	5-14	
5.4.4.2	Syntax Examples	5-14	
<b>CHAPTER</b>	<b>6</b>	<b>CLOCK SERVICES</b>	<b>6-1</b>
6.1	CLOCK SERVICE PROCESS	6-1	
6.2	SERVICE REQUEST INTERFACE	6-2	
6.2.1	Function Codes and Sequence Numbers	6-3	
6.2.2	Signal a Semaphore After a Given Time Interval (SSI)	6-4	
6.2.2.1	SSI Request Message	6-4	
6.2.2.2	SSI Reply Message	6-5	
6.2.2.3	Applications	6-5	
6.2.3	Signal a Semaphore Periodically (SSP)	6-6	
6.2.3.1	SSP Request Message	6-6	
6.2.3.2	SSP Reply Message	6-7	
6.2.3.3	Applications	6-7	
6.2.4	Cancel One or More Timer Requests (CTR)	6-7	
6.2.4.1	CTR Request Message	6-7	

6.2.4.2	CTR Reply Message	6-8	
6.2.5	Set Time of Day and Date (STD)	6-9	
6.2.5.1	STD Request Message	6-9	
6.2.5.2	STD Reply Message	6-9	
6.2.6	Get Time of Day and Date (GTD)	6-10	
6.2.6.1	GTD Request Message	6-10	
6.2.6.2	GTD Reply Message	6-10	
<b>CHAPTER</b>	<b>7</b>	<b>EXCEPTION PROCESSING</b>	<b>7-1</b>
7.1	EXCEPTION CONDITIONS, TYPES, AND SUBCODES	7-1	
7.2	DECLARING EXCEPTIONS	7-5	
7.3	EXCEPTION DISPATCHING	7-5	
7.4	EXCEPTION HANDLERS	7-7	
7.4.1	Exception-Handler Processes	7-7	
7.4.2	Exception-Handler Routines	7-8	
<b>CHAPTER</b>	<b>8</b>	<b>INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES</b>	<b>8-1</b>
8.1	DEVICE INTERRUPTS AND DEVICE-HANDLER FUNCTIONS	8-1	
8.2	THE INTERRUPT SERVICE ROUTINE	8-2	
8.3	ISRs AND INTERRUPT DISPATCHING	8-3	
8.3.1	The Interrupt Dispatch Block (IDB)	8-3	
8.3.2	Kernel Interrupt Dispatcher	8-5	
8.3.3	Establishing the Interrupt-to-ISR Interface	8-6	
8.3.3.1	Allocating IDBs and Setting Vectors	8-6	
8.3.3.2	Initializing IDBs During Start-Up	8-6	
8.3.3.3	Connecting Interrupts to ISRs	8-7	
8.4	ENTERING AND EXECUTING ISRs	8-8	
8.4.1	Entering and Executing Normal ISRs	8-8	
8.4.2	Entering and Executing Priority-7 ISRs	8-9	
8.4.3	The Fork Routine	8-9	
8.4.4	Dismissing the Interrupt	8-10	
8.5	KERNEL INTERRUPT EXIT PROCESSING	8-11	
8.6	PASCAL LANGUAGE ISR INTERFACE	8-11	
<b>CHAPTER</b>	<b>9</b>	<b>GUIDE TO WRITING A DEVICE HANDLER</b>	<b>9-1</b>
9.1	DEVICE-HANDLER OVERVIEW	9-1	
9.2	DEVICE-HANDLER PREFIX MODULE	9-3	
9.2.1	Priority Assignments	9-3	
9.2.2	DRVCF\$ Macro	9-5	
9.2.3	CTRCF\$ Macro	9-6	
9.2.4	Sample Handler Prefix Module (DYPFX.MAC)	9-9	
9.3	DEVICE-HANDLER IMPURE-AREA DEFINITION MACRO (xxISZ\$)	9-9	
9.4	DEVICE HANDLER PROPER	9-10	
9.4.1	Copyright Page	9-11	
9.4.2	Module Header	9-12	
9.4.3	Functional Description	9-12	
9.4.4	Declarations	9-12	
9.4.4.1	Local Macro Definition	9-12	
9.4.4.2	Externally Defined Symbols	9-12	
9.4.4.3	Process Definition	9-12	
9.4.4.4	Impure-Area Definition	9-14	
9.4.4.5	Pure-Area Definition	9-14	
9.4.5	Initialization Process	9-14	
9.4.6	Controller Process	9-15	
9.4.7	Interrupt Service Routine (ISR)	9-16	
9.4.8	Fork Routine	9-17	
9.4.9	Reply Subroutine	9-17	

9.4.10	Termination Procedure	9-18
9.4.11	Error-Processing Routines	9-18
9.4.11.1	Invalid Requests	9-18
9.4.11.2	Exceptions	9-19
9.4.11.3	Drive or Controller Errors	9-19
9.4.11.4	Resource Famine	9-19
9.4.11.5	Exception-Reporting Routine (\$DDEXC)	9-19
<b>APPENDIX A</b>	<b>SAMPLE DEVICE HANDLERS</b>	<b>A-1</b>
A.1	RX02 (DY) HANDLER -- SAMPLE MACRO-11 DEVICE-HANDLER SOURCE	A-2
A.2	DLV11 (XL) HANDLER -- SAMPLE MACRO-11 DEVICE-HANDLER SOURCE	A-19
A.3	DRV11 (YA) HANDLER -- SAMPLE PASCAL DEVICE-HANDLER SOURCE	A-43
<b>APPENDIX B</b>	<b>SCHEDULING HIERARCHY AND RECOMMENDED PROCESS PRIORITIES</b>	<b>B-1</b>
B.1	PRIORITY SCHEDULING HIERARCHY	B-1
B.2	RECOMMENDED PROCESS PRIORITIES	B-2
B.3	RECOMMENDED DEVICE-HANDLER PROCESS PRIORITIES	B-3
<b>APPENDIX C</b>	<b>KERNEL PRIMITIVES</b>	<b>C-1</b>
<b>APPENDIX D</b>	<b>MACRO-11 SUBROUTINE CALLING CONVENTIONS</b>	<b>D-1</b>
D.1	NORMAL MICROPOWER/PASCAL SUBROUTINE CALLING CONVENTIONS	D-1
D.2	STANDARD PDP-11 (SEQ11) SUBROUTINE CALLING CONVENTIONS	D-3
<b>APPENDIX E</b>	<b>MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE</b>	<b>E-1</b>
E.1	STRUCTURE OF A RANDOM-ACCESS DEVICE	E-1
E.1.1	Home Block	E-2
E.1.2	Directory	E-4
E.1.2.1	Directory Header	E-4
E.1.2.2	Directory Entry	E-5
E.2	DIRECTORY USE	E-7
E.2.1	Sample Directory Segment	E-7
E.2.2	Splitting a Directory Segment	E-10
E.2.3	Using RT-11 Commands to Recover Data from a Corrupted Directory	E-13
E.2.3.1	Examine Segment 1	E-13
E.2.3.2	Follow the Chain of Segments	E-14
E.2.3.3	Remove the Data from the Good Segments	E-15
E.2.3.4	Remove the Data from the Bad Segment	E-16
E.3	FILE STORAGE	E-16
E.3.1	Method	E-16
E.3.2	Size and Number of Files	E-18
E.4	RTDSP MESSAGE FORMAT	E-19
E.4.1	Requests	E-19
E.4.2	Replies	E-21
<b>APPENDIX F</b>	<b>MACRO-11 FILE SYSTEM EXAMPLE</b>	<b>F-1</b>
<b>APPENDIX G</b>	<b>LSI-11 ANALOG SYSTEM INTERFACE</b>	<b>G-1</b>
G.1	KWV11-C REAL-TIME CLOCK INTERFACE	G-3
G.1.1	Read_counts_wait	G-3

G.1.2	Read_counts_signal	G-5
G.1.3	Start_rtclock	G-7
G.1.4	Stop_rtclock	G-8
G.2	ANALOG-TO-DIGITAL (A/D) CONVERSION INTERFACE	G-8
G.2.1	Read_analog_wait	G-9
G.2.2	Read_analog_signal	G-10
G.2.3	Read_analog_continuous	G-12
G.3	DIGITAL-TO-ANALOG (D/A) CONVERSION INTERFACE: Write_analog_wait	G-13
G.4	EXAMPLES	G-14
G.4.1	Timed A/D Conversion When the KXT11-C Is Not Available	G-14.1
G.4.2	Timed D/A Conversion When the KXT11-C Is Not Available	G-16
G.4.3	Continuous A/D Conversion	G-17
G.4.4	Asynchronous, Block-Mode A/D Conversion	G-18
<b>APPENDIX H</b>	<b>KXT11-C PERIPHERAL PROCESSOR INTERFACE</b>	<b>H-1</b>
H.1	ARBITER INTERFACE TO THE KXT11-C	H-2
H.1.1	Sending Data to the KXT11-C	H-3
H.1.2	Receiving Data from the KXT11-C	H-4
H.2	KXT11-C INTERFACE TO THE ARBITER	H-5
H.2.1	Receiving Data from the LSI-11 Bus	H-6
H.2.2	Sending Data to the LSI-11 Bus	H-6
H.3	KXT11-C TU58 INTERFACE	H-7
H.4	KXT11-C DMA TRANSFER CONTROLLER INTERFACE	H-7
H.4.1	DMA Transfer	H-10
H.4.2	DMA Search	H-10
H.4.3	DMA Transfer While Search	H-11
H.4.4	DMA Channel Allocation	H-12
H.4.5	DMA Channel Deallocation	H-12
H.4.6	DMA Status Return	H-13
H.4.7	DMA Function Call Example	H-13
H.5	KXT11-C ASYNCHRONOUS SERIAL LINE INTERFACE	H-14
H.6	KXT11-C SYNCHRONOUS SERIAL LINE INTERFACE	H-14
H.7	KXT11-C PARALLEL PORT AND TIMER/COUNTER INTERFACE	H-15
H.7.1	Reading from a PIO Port	H-16
H.7.2	Writing to a PIO Port	H-17
H.7.3	Pattern Recognition	H-18
H.7.4	PIO DMA Process	H-20
H.7.5	Timer/Counter Commands	H-20
H.7.5.1	Timer Setup	H-20
H.7.5.2	Reading a Timer	H-21
H.7.5.3	Clearing a Timer	H-22
H.7.5.4	Notes on Timer Requests	H-22
H.8	LINE-FREQUENCY CLOCK INTERFACE	H-22
H.9	LOAD APPLICATION ONTO KXT11-C PROCEDURE	H-22
H.9.1	MIM File	H-23
H.9.2	User's Interface	H-23
H.9.3	Program Example	H-24

INDEX

Index-1

## FIGURES

<b>FIGURE</b>	<b>2-1</b>	<b>Process State Transitions</b>	<b>2-7</b>
	2-2	Process Control Block (PCB)	2-13
	2-3	ROM/RAM Memory Layout	2-17
	2-4	RAM-Only Memory Layout	2-17
	2-5	Kernel Mapping	2-20

2-6	General-Process Mapping	2-21
2-7	Device-Access Process Mapping	2-21
2-8	Driver Process Mapping	2-22
2-9	Privileged-Process Mapping	2-22
2-10	Interrupt Service Routine Mapping	2-23
2-11	System Queue Structures	2-32
2-12	Free-Memory Pool	2-34
3-1	SEND\$/\$NDC\$ Packet Format	3-83
4-1	I/O Request Packet Format	4-6
4-2	Format of Send Buffer for an I/O Request	4-11
4-3	Format of an I/O Reply Message	4-13
4-4	General X.25 Frame Format	4-81
4-5	Command Byte Formats	4-82
5-1	File Identification Block Fields	5-11
7-1	Kernel Exception Processing	7-7
E-1	Random-Access Device	E-2
E-2	Format of Home Block	E-3
E-3	Format of Device Directory	E-4
E-4	Format of Directory Entry	E-5
E-5	Format of Status Word	E-5
E-6	Format of Date Word	E-7
E-7	Directory Listings	E-8
E-8	Directory Segment	E-9
E-9	Storing a New File	E-10
E-10	Full Directory Segment	E-11
E-11	Directory Before Splitting	E-12
E-12	Directory After Splitting	E-12
E-13	Directory Links	E-13
E-14	Worksheet for a Directory Chain with Four Segments	E-14
E-15	Worksheet for a Directory Chain with Nine Segments	E-15
E-16	Random-Access Device with Two Permanent Files	E-17
E-17	Random-Access Device with One Tentative File	E-17
E-18	Random-Access Device with Two Tentative Files	E-17
E-19	Random-Access Device with Four Permanent Files	E-18
E-20	RTDSP Request Message Format	E-20
E-21	RTDSP Reply Message Format	E-21

## TABLES

TABLE	4-1	Standard Device Handlers and Device Identifiers	4-1
	4-2	PDP-11 Request Queue Names, Prefix Files, and Hardware	4-4
	4-3	KXT11-C Request Queue Names, Prefix Files, and Hardware	4-4
	4-4	Two-Port RAM Data Channel Addresses	4-56
	4-5	KK Prefix File Defaults	4-56
	7-1	Exception Types and Subcodes	7-2
	9-1	Recommended Device-Handler Process Priorities	9-4
	E-1	Contents of Home Block	E-3

## PREFACE

### Manual Objectives and Reader Assumptions

This manual describes the organization of the MicroPower/Pascal runtime system and the services it provides for user programs. The content of this manual is based on the assumption that you have read the Introduction to MicroPower/Pascal. In addition, it is assumed that you are familiar with either Pascal or MACRO-II. All MicroPower/Pascal microcomputer software development is done with one or both of the specified development languages. Additional runtime services reference information for writing applications in Pascal is contained in the MicroPower/Pascal Language Guide.

### Document Structure

Nine chapters comprise this manual, as follows:

- Chapter 1 presents an overview of the MicroPower/Pascal runtime system. Kernel organization is described in general terms, including an overview of primitive services and system processes.
- Chapter 2 describes MicroPower/Pascal processes and system data structures. You must read and understand that information before attempting to write application code using the runtime services described in the remainder of this manual.
- Chapter 3 gives detailed descriptions for each of the MACRO-II primitive service requests. Primitive services provided by the MicroPower/Pascal kernel are also accessible to applications written in Pascal. Refer to the MicroPower/Pascal Language Guide for details on issuing kernel primitive requests in Pascal programs.
- Chapter 4 provides complete user information for the standard device handlers supplied in the MicroPower/Pascal kit. Information is provided for both Pascal and MACRO-II users.
- Chapter 5 describes the file system services that are available to the MACRO-II user. The Pascal interface to the file system is described in the MicroPower/Pascal Language Guide. Files produced by the file system are media-compatible with RT-II.

- Chapter 6 presents information for using the MicroPower Pascal clock services, available only for target systems that contain time-of-day clock (TOD) or real-time-clock hardware.
- Chapter 7 describes MicroPower/Pascal exception processing. (Exceptions are hardware or software errors or traps that occur only when application programs are executed or delayed on the target system in the real-time environment.)
- Chapter 8 describes kernel interrupt distribution, interrupt service routines (ISRS), and fork routines.
- Chapter 9 presents guidelines for writing a MicroPower Pascal device handler for nonstandard hardware devices -- devices not supported by handlers supplied in your MicroPower Pascal distribution kit. This chapter describes the necessary components of a device handler and the handler's interface to the application program and refers to sample handlers written in both Pascal and MACRO-11.

#### **Associated Documents**

The following software documentation is required for complete reference purposes. Refer to the documentation list for your host operating system.

- **RT-11 Host:**
  1. MicroPower/Pascal-RT documentation set. A complete list of documents is contained in the MicroPower Pascal-RT Documentation Directory (Order No. AA-W0660-TD).
  2. RT-11 V5 host operating system documentation set. A subset of the RT-11 V5 documentation set is contained in the MicroPower Pascal-RT documentation set. No additional RT-11 documentation is required for MicroPower Pascal-RT application software development.
- **RSX-11M/M-PLUS Host:**
  1. MicroPower/Pascal-PSX documentation set. A complete list of documents is contained in the MicroPower Pascal-PSX Installation Guide (Order No. AA-AK10A-TC).
  2. RSX-11M/M-PLUS host operating system documentation set. Refer to the documentation set supplied with your host operating system.
- **VAX/VMS Host:**
  1. MicroPower/Pascal-VMS documentation set. A complete list of documents is contained in the MicroPower/Pascal-VMS Installation Guide (Order No. AA-AT16A-TE).
  2. VAX/VMS host operating system documentation set. Refer to the documentation set supplied with your host operating system.

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

1. Microcomputer handbooks
  - Microcomputers and Memories (Order No. EB-20912-20)
  - Microcomputer Interfaces Handbook (Order No. EB-23144-18)
2. SBC-11/21 Single Board Computer User's Guide (Order No. EK-KXT11-UG) (required when developing SBC-11/21 applications)
3. DPV11 Serial Synchronous Interface Technical Manual (Order No. EK-DPV11-TM) (required when developing applications using DPV11 communications hardware)
4. LSI-11 Analog System User's Guide (Order No. EK-AXV11-46-002) (required when developing applications using the SDV11-C, AAV11-C, AXV11-C, or KWV11-C analog I/O boards)
5. MSCP Basic Disk Functions Manual (Order No. AA-L619A-TK) (required when developing applications using MSCP disk-class devices)
6. KXT11-CA User's Guide (Order No. EK-KXTCA-UG) (required when developing KXT11-C applications)
7. Additional hardware documentation for microcomputer hardware presently not covered in the microcomputer handbooks

#### Document Conventions

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 parameters (or other syntax elements) that you must include for your particular application.
2. Optional parameters and syntax are shown within brackets ([ ]). This document convention is used mainly in Chapter 3 for kernel primitive parameters. Before considering any parameters optional, carefully read Chapter 3, Section 3.1.1, which describes the general form and usage rules for the prim\$ macro variant.
3. In this manual, some MACRO-11 syntax examples are shown with long macro invocations continued on a second line -- for example, the CRPCS and DFSPCS macro calls. However, when writing source code in MACRO-11, you must have each long macro invocation on a single line.

#### Symbols

In this manual, the numeric values for symbols for data structure sizes, offsets, and so forth, are subject to change. Therefore, use symbol names rather than numeric values for system data structure components.

## CHAPTER 1

### INTRODUCTION

This manual describes the organization of the MicroPower/Pascal runtime system and the services it provides for user programs. The explicit, user-requested services include real-time kernel primitive operations (Chapter 3), standard device-I/O services (Chapter 4), file system services (Chapter 5), and clock services (Chapter 6). Implicit services include process scheduling (Chapter 2), trap/exception processing (Chapter 7), and interrupt dispatching (Chapter 8). Chapter 2 provides an overview of the process/kernel relationship. It discusses the dynamic characteristics of a MicroPower/Pascal concurrent process and gives a detailed description of process states, scheduling, and the effect of process-mapping type in a mapped environment. Chapter 2 also describes the system data structures the kernel uses to implement primitive operations. Chapter 9 is a guide for writing device handlers for custom hardware devices.

Other manuals in the MicroPower/Pascal documentation set focus on the Pascal user and provide only Pascal-oriented descriptions. Much of the information in this manual is applicable to both Pascal and MACRO-11 users; wherever possible, concepts are explained in terms of both Pascal and MACRO language constructs.

Some of the information, however -- particularly Chapter 3, which describes the MACRO-11 primitive service requests -- is pertinent only to MACRO-11 programmers. Analogous information for Pascal programmers is provided primarily in Part Two of the MicroPower/Pascal Language Guide.

#### 1.1 THE MICROPOWER/PASCAL RUNTIME SYSTEM

The MicroPower/Pascal runtime system is the collection of DIGITAL-supplied software that resides in the target system and provides the execution-time environment for application programs. The runtime system consists of the MicroPower/Pascal kernel and a number of system-level processes. The kernel provides the set of basic operations, called primitives, that are required for concurrent programming. These primitives implement process creation and deletion, process synchronization, and interprocess communication, for example.

User programs obtain primitive services by invoking appropriate kernel routines through a service request interface provided for both Pascal and MACRO-11 programming. The kernel also performs implicit functions such as process scheduling, interrupt dispatching, and trap/exception dispatching; these functions are largely transparent to user programs. The kernel is modular; when you build the application, you can tailor the kernel to match both the target hardware configuration and the primitive service requirements of the application processes.

## INTRODUCTION

The DIGITAL supplied system processes provide device-handling services for commonly used I/O devices and device interfaces and basic clock services. Application processes obtain these services by using queue semaphore primitives to send request messages to the appropriate system process. The system processes are included in the target system during system building on an individual, as-needed basis.

### 1.2 KERNEL ORGANIZATION

The highly structured MicroPower/Pascal kernel consists of many small program modules with well-defined functions and interfaces. This not only makes the kernel easier to configure, maintain, and modify, but also allows a great deal of common code to be used in kernels for different hardware environments -- for example, mapped versus unmapped systems. (The common code contributes significantly to kernel reliability.) Among the many kernel modules, however, six major functional components can be distinguished:

1. The primitive service routines, the 38 modules that implement the primitive operations requested by processes
2. The primitive dispatcher, which receives all primitive service requests and passes control to the appropriate primitive service module
3. The interrupt dispatcher, which receives all device interrupts and passes control to appropriate service routines, providing the necessary entry and exit processing
4. The exception dispatcher, which receives all exception conditions -- actual and simulated processor traps -- and transfers control as required for handling of the exception
5. The scheduler, which allocates the CPU to processes, according to priority, on an event-driven, preemptive basis
6. The system-initialization routine, which initializes kernel data structures and installs static processes at start-up/restart time

The 38 primitive service modules form the largest kernel component. This component is configurable, however; only those primitives used in a given application system need to be included in the kernel for that system. The remaining components, along with other miscellaneous functions and common kernel subroutines, constitute the mandatory kernel core.

#### 1.2.1 Overview of Primitive Services

The primitive service component supplies 38 primitive operations for concurrent programming. These primitives can be grouped into seven categories, as follows:

1. Process management: primitives that include create, delete, suspend, and resume processes and force termination.
2. Resource management: primitives that permit creation and deletion of system data structures -- semaphores and ring buffers -- and allocation and deallocation of message packets.

## INTRODUCTION

3. Process synchronization: primitives that synchronize cooperating processes by performing signal and wait operations on binary and counting semaphores.
4. Message transmission and synchronization: primitives that perform interprocess communication by operating on queue semaphores and by combining packet queuing and dequeuing with signal and wait operations.
5. Ring buffer management: primitives that facilitate variable-length data transfers between processes, via ring buffers, without the need for close synchronization between putter and getter.
6. Exception management: primitives that direct the dispatching of hardware and software exception conditions to an appropriate exception-handling process or exception service routine and permit a process to report a software exception. (The hardware-detected events reported by processor traps other than IOT or power-fail constitute the MicroPower/Pascal hardware exceptions.)
7. Interrupt management: primitives that affect interrupt dispatching; used only by processes that manage an I/O or clock device.

Primitives are described briefly in the following subsections. Chapter 3 contains complete descriptions for the MACRO-11 programmer. Refer to Part Two of the MicroPower/Pascal Language Guide for a description of the Pascal primitive service requests. (Several process-management services are transparent, or implicit, in Pascal programming; the primitives are invoked automatically when required, rather than by explicit service requests. These few differences in MACRO and Pascal usage are indicated in the next subsection.)

### NOTE

Several assembly-time macros -- Define Static Process (DFSPCS), Define a Pure Program Instruction Section (PURE\$), Define a Pure Program Data Section (PDATS), and Define an Impure Program Data Section (IMPUR\$) -- are defined in Chapter 3 for MACRO-11 programming convenience. Chapter 3 also describes two special kernel services, used only in interrupt service routines, that are not implemented as primitive operations. The two kernel services are Fork Processing (FORK\$) and Drop CPU Priority (P7SYSS).

#### 1.2.1.1 Process-Management Primitives - These eight primitives are the following:

1. Create Process: allows an existing process to dynamically create a new process and cause it to be scheduled for execution. In Pascal, invocation of this primitive is implicit in a process-invocation statement.
  - MACRO-11 service request name: CRPCS
  - Pascal equivalent: Process invocation statement

## INTRODUCTION

2. Delete Process: allows a process to delete itself from the system. (This is the only way in which a process can terminate.) In Pascal, invocation of this primitive is implicit if control flow reaches the end of the level-0 block for a static process or the end of a PROCESS declaration block for a dynamic process.
  - MACRO-II service request name: DLPCS
  - Pascal equivalent: None
3. Suspend Process: allows a process to suspend another active process or itself. Once suspended, a process remains in that state, ineligible for execution, until it is resumed by another process.
  - MACRO-II service request name: SPNDS
  - Pascal equivalent: SUSPEND function
4. Resume Process: allows a process to reactivate another, suspended process.
  - MACRO-II service request name: RSUMS
  - Pascal equivalent: RESUME function
5. Stop Process: allows one process to force another process to execute its termination routine or (Pascal) TERMINATE procedure. (The stopped process must delete itself in order to go away.)
  - MACRO-II service request name: STPCS
  - Pascal equivalent: STOP procedure
6. Get Process Status: allows one process to obtain information about the status of either itself or another process.
  - MACRO-II service request name: GTSTS
  - Pascal equivalent: GET\_STATUS procedure
7. Change Process Priority: allows a process to modify its scheduling priority. Normally, this primitive is used to lower priority from a very high start-up value used only for initialization. (In Pascal, the INITIALIZE procedure attribute indirectly serves this purpose.)
  - MACRO-II service request name: CHGPs
  - Pascal equivalent: CHANGE\_PRIORITY procedure
8. Schedule Process: allows a process to relinquish control of the CPU to another process of equal priority, if one is ready to execute.
  - MACRO-II service request name: SCHDS
  - Pascal equivalent: SCHEDULE procedure

## INTRODUCTION

1.2.1.2 Resource-Management Primitives - These six primitives are the following:

1. Create Structure: creates a system data structure -- a semaphore, ring buffer, or unformatted structure -- in kernel data space.
  - MACRO-II service request name: CRSTS
  - Pascal equivalents: CREATE\_BINARY\_SEMAPHORE function  
CREATE\_COUNTING\_SEMAPHORE function  
CREATE\_QUEUE\_SEMAPHORE function  
CREATE\_RING\_BUFFER function
2. Delete Structure: deletes a system data structure.
  - MACRO-II service request name: DLSTS
  - Pascal equivalent: DESTROY procedure
3. Get Structure Value: obtains the characteristics -- for example, type -- and value of a system data structure.
  - MACRO-II service request name: GVALS
  - Pascal equivalent: GET\_VALUE procedure
4. Allocate Packet: obtains an empty message packet from the kernel's free-packet pool (returns a pointer).
  - MACRO-II service request name: ALPKS
  - Pascal equivalent: ALLOCATE\_PACKET procedure
5. Conditionally Allocate Packet: obtains an empty message packet from the kernel's free-packet pool but does not block the process if no packets are available.
  - MACRO-II service request name: ALPCS
  - Pascal equivalent: COND\_ALLOCATE\_PACKET function
6. Deallocate Packet: returns a message packet to the kernel's free-packet pool, thus freeing the packet for reuse.
  - MACRO-II service request name: DAPKS
  - Pascal equivalent: DEALLOCATE\_PACKET procedure

1.2.1.3 Process-Synchronization Primitives - These five primitives operate on either a binary or a counting semaphore and are used by two or more cooperating processes for mutual exclusion and other forms of synchronization. A binary semaphore is a variable that can assume the values of 0 and 1. The two basic operations defined on a binary semaphore are:

SIGNAL(B): If B = 1 then B := B + 1

WAIT(B): If B = 1 then B := B - 1  
else  
    Process must wait  
    'becomes "blocked"'  
    until B = 1, then  
    B := B - 1

## INTRODUCTION

A signal of a binary semaphore having a value of 0 will allow one subsequent wait to proceed without blocking the process issuing the wait. Signaling a binary semaphore having a value of 1 has no effect; one process issuing a subsequent wait will proceed without blocking.

A counting semaphore uses a variable that can assume a value greater than 1. The two basic operations defined on a counting (C) semaphore are:

```
SIGNAL(C): C := C + 1  
  
WAIT(C): If C > 0 then C := C - 1  
          else  
          Process must wait  
          until C > 0, then  
          C := C - 1
```

As with binary semaphores, a signal of a counting semaphore having a value of 0 will allow one subsequent wait operation to proceed without blocking the process. Unlike binary semaphores, however, successive signals without intervening wait operations are not lost. Each signal is counted and will allow one wait to proceed without blocking.

The process-synchronization primitives are the following:

1. Signal Semaphore: performs an unconditional signal operation on a specified semaphore.
  - MACRO-11 service request name: SGNL\$
  - Pascal equivalent: SIGNAL procedure
2. Wait on Semaphore: performs an unconditional wait operation on a specified semaphore.
  - MACRO-11 service request name: WAITS\$
  - Pascal equivalent: WAIT procedure
3. Conditionally Signal Semaphore: performs a conditional signal operation, which increments the semaphore variable only if a process is already waiting on the semaphore. The primitive returns a FALSE indication if the signal was not performed.
  - MACRO-11 service request name: SGD\$
  - Pascal equivalent: COND\_SIGNAL function
4. Conditionally Wait on Semaphore: performs a conditional wait operation, which decrements the semaphore variable only if the semaphore has already been signaled (that is, its value is nonzero). This test-semaphore-and-decrement-if-possible operation never causes the requesting process to block. The primitive returns a FALSE indication if the wait was not performed.
  - MACRO-11 service request name: WAIT\$
  - Pascal equivalent: COND\_WAIT function
5. Signal All Waiting Processes: performs a special form of signal operation, which unlocks any and all processes that may be waiting on the specified semaphore and sets the semaphore value to 1 unconditionally.
  - MACRO-11 service request name: SALL\$
  - Pascal equivalent: SIGNAL\_ALL procedure

## INTRODUCTION

1.2.1.4 **Message-Transmission Plus Synchronization Primitives** - These eight primitives operate on queue semaphores and combine message-packet transmission with signal and wait operations. A queue semaphore is a generalization of the counting semaphore and has a queue of elements associated with it, in addition to the counter variable. (A standard MicroPower/Pascal queue element is called a packet.)

The basic signal-queue-semaphore operation adds a packet to the queue and increments the counter variable. The basic wait-on-queue-semaphore operation removes a packet, if any, from the queue and decrements the variable; if the queue is empty, the process must wait until an element can be removed. Thus, the value of the counter variable always represents the number of elements on the queue. The synchronization characteristics of queue semaphores are identical to those of counting semaphores.

Two distinct levels of queue semaphore operations are supplied, one built on the other. The higher-level, more automatic operations (send and receive) are provided specifically for general processes in a mapped-memory environment. They can, however, be used by any process in either a mapped or an unmapped environment. The individual queue semaphore primitives, beginning with the lower-level operations, are the following:

1. **Signal Queue Semaphore, or Put Packet:** signals the specified semaphore and places a packet pointer -- supplied by the caller -- on the semaphore's packet queue.
  - MACRO-11 service request name: SGQS\$
  - Pascal equivalent: PUT\_PACKET procedure
2. **Wait on Queue Semaphore, or Get Packet:** performs a wait operation on the specified semaphore by removing a packet pointer from the queue and returning it to the requesting process if a packet is available immediately. If not, the process blocks until the semaphore is signaled.
  - MACRO 1 service request name: WAIQS
  - Pascal equivalent: GET\_PACKET procedure
3. **Conditionally Signal Queue Semaphore:** performs a conditional signal-queue operation, which places a packet pointer -- supplied by the caller -- on the semaphore's queue only if a process is already waiting for a packet on that semaphore. The primitive returns a FALSE indication if the signal operation was not performed.
  - MACRO-11 service request name: SGQS\$
  - Pascal equivalent: COND\_PUT\_PACKET function
4. **Conditionally Wait on Queue Semaphore:** performs a conditional wait-queue operation, which removes a packet pointer from the semaphore's queue and returns it to the requestor only if a packet is on the queue (that is, the semaphore had already been signaled). This test-semaphore-and-get-packet-if-possible operation never causes the requesting process to block. The primitive returns a FALSE indication if a packet was not immediately available.
  - MACRO-11 service request name: WAQS\$
  - Pascal equivalent: COND\_GET\_PACKET function

## INTRODUCTION

5. Send Data via queue semaphore: allocates a packet -- obtains a free packet from the pool -- copies caller-specified data into the packet, and then performs a signal operation on the specified queue semaphore. (See the Allocate Packet description for possible blocking condition.)
  - MACRO-11 service request name: SEND\$
  - Pascal equivalent: SEND procedure
  - Pascal variant: SEND\_ACK procedure
6. Receive Data via queue semaphore: performs a wait operation on a specified queue semaphore, then copies data from the packet thus obtained into a caller-specified data area, and finally deallocates the packet (that is, returns the packet to the free-packet pool). The calling process will block if a packet is not immediately available.
  - MACRO-11 service request name: RCVDS\$
  - Pascal equivalent: RECEIVE procedure
  - Pascal variant: RECEIVE\_ACK procedure
7. Conditionally Send Data: performs a send-data operation as described above, but only if a process is already waiting to get a packet or receive packet data via the specified queue semaphore. The primitive returns a FALSE indication if the send operation was not performed.
  - MACRO-11 service request name: SNDC\$
  - Pascal equivalent: COND\_SEND procedure
  - Pascal variant: COND\_SEND\_ACK procedure
8. Conditionally Receive Data: performs a receive-data operation as described above, but only if a packet is on the specified semaphore's queue. This test-semaphore-and-receive-data-if-available operation never causes the requesting process to block. The primitive returns a FALSE indication if the receive operation was not performed.
  - MACRO-11 service request name: RCVCS\$
  - Pascal equivalent: COND\_RECEIVE procedure
  - Pascal variant: COND\_RECEIVE\_ACK procedure

Note that in a mapped environment, a process must have privileged mapping to fully use the lower-level queue semaphore primitives (Put Packet and Get Packet). Packets reside in kernel data space, and the process must be mapped to that space in order to access -- write into or read from -- the packet.

**1.2.1.5 Ring Buffer Primitives** - These five primitives operate on ring buffer structures, which facilitate variable-length data transfers, normally of byte data, between processes, without the need for tight, signal/wait synchronization between them. The size, or

## INTRODUCTION

capacity, of a ring buffer is determined when the structure is created; this size can range from 8 bytes to approximately 8K bytes. The ring buffer primitives are the following:

1. Get Element: moves a specified number of bytes of data from a ring buffer to a data area specified by the requestor. If the buffer does not contain enough data to satisfy the request, the calling process blocks until a sufficient amount of data is put into the buffer by another process.
  - MACRO-11 service request name: GELMS
  - Pascal equivalent: GET\_ELEMENT procedure
2. Put Element: moves a specified number of bytes of data from a data area specified by the requestor to the ring buffer. If the buffer has insufficient space to accommodate the new element, the calling process blocks until sufficient space becomes available (due to subsequent Get operations).
  - MACRO-11 service request name: PELMS
  - Pascal equivalent: PUT\_ELEMENT procedure
3. Conditionally Get Element: gets a data element of specified length from a ring buffer if the buffer contains enough data to satisfy the request. This primitive will not cause the calling process to block. If the buffer does not contain enough data to satisfy the request, the primitive either gets as many bytes as possible or moves no data at all, depending on the output mode -- stream or record -- specified for the buffer when it was created. This primitive returns a value indicating the number of bytes that remain to be moved following the operation.
  - MACRO-11 service request name: GELCS
  - Pascal equivalent: COND\_GET\_ELEMENT function
4. Conditionally Put Element: puts a data element of specified length into a ring buffer if the buffer has enough space to accommodate the element. This primitive will not cause the calling process to block. If the buffer does not have enough space to accommodate the entire element, the primitive either puts as many bytes as possible or moves no data at all, depending on the input mode -- stream or record -- specified for the buffer when it was created. This primitive returns a value indicating the number of bytes that remain to be moved following the operation.
  - MACRO-11 service request name: PELCS
  - Pascal equivalent: COND\_PUT\_ELEMENT function
5. Reset Ring Buffer: empties a specified ring buffer of all data.
  - MACRO-11 service request name: RBUFS
  - Pascal equivalent: RESET\_RING\_BUFFER procedure

## INTRODUCTION

1.2.1.6 Exception-Processing Primitives - These four primitives are the following:

1. Connect to Exception Condition: allows a process to establish itself as an exception handler for processes that belong to a given exception-handling group.
  - MACRO-11 service request name: CCNDS
  - Pascal equivalents: CONNECT\_EXCEPTION procedure  
DISCONNECT\_EXCEPTION procedure
2. Dismiss Exception Condition: allows an exception-handler process to dismiss an exception, releasing the faulting process from exception wait state for further disposition by the kernel.
  - MACRO-11 service request name: DEXCS
  - Pascal equivalent: RELEASE\_EXCEPTION procedure
3. Set Exception Routine Address: allows any process to specify the entry point of an internal exception service routine or procedure that will handle exceptions caused by the process.
  - MACRO-11 service request name: SERAS
  - Pascal equivalents: ESTABLISH procedure  
REVERT procedure
4. Report Exception: allows a process to report a software exception condition or to force a hardware exception (simulate a processor trap).
  - MACRO-11 service request name: REXCS
  - Pascal equivalent: REPORT procedure

1.2.1.7 Interrupt-Management Primitives - These two primitive operations involve interrupt service routines (ISRs). These primitives are the following:

1. Connect to Interrupt: allows a device-handling process to connect an ISR to a specified interrupt vector. (A Pascal variant of this primitive allows a process to indirectly connect a binary or a counting semaphore to an interrupt vector.)
  - MACRO-11 service request name: CINTS
  - Pascal equivalent: CONNECT\_INTERRUPT procedure
  - Pascal variant: CONNECT\_SEMAPHORE procedure
2. Disconnect from Interrupt: allows a device-handling process to disconnect an ISR from a specified interrupt vector.
  - MACRO-11 service request name: DINTS
  - Pascal equivalent: DISCONNECT\_INTERRUPT procedure
  - Pascal variant: DISCONNECT\_SEMAPHORE procedure

## INTRODUCTION

### 1.2.2 Overview of System Processes

System processes provide commonly used hardware-oriented system services available to user programs. System processes include standard (DIGITAL-supplied) device handlers and the clock process. Chapters 4 and 6, respectively, describe those system processes.

**1.2.2.1 Standard Device Handlers** - Eighteen standard device handlers are included in MicroPower/Pascal. They include device handlers (also called device drivers) for hardware devices, as follows:

Device Handler	Device Hardware Supported
AA	ADV11-C and AXV11-C analog input converters
DD	TU58 DECTape II cartridge tape drive
DL	RLP1 and RLP2 disks (RLV11, RLV12, and RLV21 controllers)
DU	MSCP disk-class devices, including RDP1 and RXF1 (RQDX1 controller)
DY	PX82/RXV21 floppy disk
KW	KWV11-C programmable real-time clock
KX	KXT11-C Two-Port RAM (arbiter side)
XA	DRV11-I high-density parallel line interface
XL	DLV11, DLV11-E, DLV11-F, DLV11-J, MXV11-A, MXV11-B, and SBC-11/21 serial line unit (SLU) interfaces
XP	DPV11 communications line
YA	DRV11 parallel line unit
YF	SBC-11/21 parallel I/O (PIO) port
DD (KXT11-C)	TU58 DECTape II cartridge tape drive
KK	KXT11-C Two-Port RAM (KXT11-C side)
OD	KXT11-C DMA Transfer Controller
XL (KXT11-C)	KXT11-C asynchronous serial line
XS	KXT11-C synchronous serial line
YK	KXT11-C parallel port and timer counter

A device handler is a process, or a family of cooperating processes, that accepts requests for device-level I/O operations from other processes. Device handlers communicate and synchronize with other processes in the application by using standard primitive operations. I/O service requests for a particular hardware device are passed to the device handler in the form of a request message (packet). Each handler maintains a request queue through which device-handler I/O requests are passed. After receiving a request, the device handler

## INTRODUCTION

performs all device-level, interrupt-level, and fork-level processing for the requesting process. When the I/O operation has been completed, the device handler signals the requesting process and returns completion status via a reply message packet. The reply message packet indicates successful completion or error, including severity and type, as applicable, and the number of bytes successfully transferred.

Standard I/O functions generally supported by device handlers include read (physical and logical), write (physical and logical), set device characteristics, and get device characteristics. Other device-specific functions are supported for each device. (Refer to individual device-handler descriptions in Chapter 4 for details.)

Device handlers can be written in either Pascal or MACRO-11, and device-handler processes can be accessed by processes written in either Pascal or MACRO-11. Of the 18 standard device handlers included in the MicroPower Pascal software, the AA, PW, YA, and YC handlers are written in Pascal; all others are written in MACRO-11.

**1.2.2.2 Clock Process** - The clock process maintains system time and date and provides basic timer and clock services to other processes in the application system. All clock-process functions are based on hardware-generated clock interrupts that usually occur at the power line frequency (60 Hz or 50 Hz). Other clock frequency rates can be configured for specific user applications.

Five clock functions permit setting and getting the time of day and date, signaling a semaphore either periodically or after a given time interval, or canceling a signal's semaphore request. These functions are the following:

1. Signal a Semaphore After a Given Time Interval: signals the requestor's semaphore upon expiration of a specified time interval. (The time interval is expressed as a number of clock ticks.)
2. Signal a Semaphore Periodically: signals the requestor's semaphore periodically until the request is canceled. (The time intervals for all signal operations are equal and are expressed as a number of clock ticks.)
3. Cancel One or More Timer Requests: cancels one or more previously issued requests to either Signal a Semaphore After a Given Time or Signal a Semaphore Periodically.
4. Set Time of Day: sets both the time of day and the date to specified values (second, minute, hour, day, month, year). (The requesting process will normally obtain current values from an operator before issuing this request.)
5. Get Time of Day and Date: returns to the requestor a reply message containing the time-of-day and date values. In addition, the reply message includes the number of elapsed ticks in the current second and the clock tick frequency (for example, the power line frequency).

## CHAPTER 2

### PROCESSES AND SYSTEM DATA STRUCTURES

This chapter begins with a general description of processes and then presents pertinent implementation-related details. Later sections describe the significant data structures defined by the MicroPower/Pascal kernel. Some of the information in this chapter is provided primarily for debugging purposes.

#### 2.1 PROCESSES

A MicroPower/Pascal process is an independent, asynchronous CPU activity, or task. Process execution proceeds concurrently -- logically in parallel -- with the execution of other processes in a given application. (The basic characteristics of a MicroPower/Pascal process are the same as those described for a concurrent process or a parallel process in the recent literature on concurrent programming.) The kernel's event-driven scheduling mechanism provides each process with its own virtual CPU (in a single-processor environment). Thus, a process can be thought of as a sequential program that can communicate and interact with other such programs executing in parallel on separate virtual processors to achieve a common goal. This goal might be, for instance, to monitor and control several related aspects of a particular real-time environment.

Since the actual CPU is shared by processes on an event-triggered basis (as opposed to equal-interval time slicing), the execution rate of one process relative to another is generally unpredictable, particularly among processes of the same scheduling priority. The MicroPower/Pascal process-synchronization primitives, however, allow functionally related processes to execute in proper time relationship.

One source program can define many processes, as described in Section 2.1.1. Since all the processes so defined exist in the same virtual address space, they can access shared data directly and can use common subroutines or procedures. Again, proper use of MicroPower/Pascal synchronization primitives permits several processes to modify shared data in a safe, controlled fashion. Also, multiple processes can be based on one (reentrant) instruction sequence, with a unique data area for each process.

The process construct allows you to decompose an otherwise monolithic sequential program into any number of autonomous subprograms that are scheduled independently when triggered by appropriate events. These events may be external, as signaled by a device interrupt, or internal, as signaled by another process (for example, availability of a shared resource or data item), and generally are a mixture of the two. The process approach avoids the wasteful busy-waiting loops that would otherwise be needed to synchronize with critical device

## PROCESSES AND SYSTEM DATA STRUCTURES

interrupts. This allows more efficient use of the CPU and other hardware resources and a more flexible response to multiple external events of varying urgency.

The process construct also provides a simpler conceptual approach to solving many real-time problems. For example, consider an application involving a windowed display; the physical display screen is divided into a number of subareas, or windows. Each window is to be a virtual display that is updated independently in response to some set of external events. A sequential programming approach would require a complicated screen-management algorithm to ensure complete and valid updating of each part of the screen, assuming that the triggering events are asynchronous. MicroPower/Pascal allows the programmer to manage each window with a separate process and to assign priorities to the processes on the basis of the relative importance or timeliness of the data to be displayed in each window. Programming a windowed display then becomes conceptually straightforward.

A process is essentially a dynamic, execution-time entity. At execution time, a process consists of the following:

- A block of control information (process control block), created and maintained by the kernel, that reflects the context of the process at any given point. This block of information exists only during the lifetime of the process it describes and is the "activation record" of the process.
- An instruction sequence or "procedure" that the process executes. (This instruction sequence is usually in the form of a closed loop.) The instruction sequence associated with a process is identified in the process's context simply by the address to which control is to be transferred when the CPU is next dispatched to the process.
- A set of data segments (process stack, local variables, and so forth) that are unique to the process, plus any shared data.

Note particularly that an instruction sequence, if reentrant, may be shared (concurrently executed) by a number of processes. Thus, a process represents one specific invocation of an instruction sequence -- usually a loop -- as an independent scheduling unit. The process control block maintains a continuous record of the context and the "activation status" of that scheduling unit, as described in Section 2.1.5.

### 2.1.1 Static and Dynamic Processes

A static process is one of the processes known to the kernel at system-initialization time and is always present after power-on or system-reset processing is completed. The kernel's initialization (INIT) procedure creates a process control block for and schedules each static process.

In Pascal, a static process is implicitly defined by a [SYSTEM(MicroPower),...] PROGRAM declaration. (Other optional attributes within the brackets specify characteristics such as stack/heap size, mapping type, and priority.) The main body of the program, together with all procedures and functions called from main level, constitute the instruction segments associated with the static process. Likewise, the variables declared at main level, together with the stack space allocated to the main program, constitute the data segments associated with the static process.

## PROCESSES AND SYSTEM DATA STRUCTURES

A procedure declared at the outermost level with the [INITIALIZE] attribute has a special relationship to the static process and has a special characteristic relative to all other Pascal static processes in the application system. If an [INITIALIZE] PROCEDURE declaration exists in a given program, it is executed automatically before the corresponding static process (main program body) is initially executed. Furthermore, this initialization procedure has an implicit scheduling priority of 255, the highest possible priority value. (User-specified priority values may not exceed 254.) This guarantees not only that the initialization procedure will run before its associated static process, but also that it will run before any other Pascal process executes.

The purpose of the initialization procedure is to permit creation of any system data structures -- semaphores or ring buffers -- that other processes depend on for proper operation, before any such process can attempt an operation on the structure. For example, an initialization procedure might create a queue semaphore on which other processes will perform a Send operation to request a service. This avoids the potential race conditions that could arise if one process were to depend on another to run first. (Relative process priorities should not be relied on, and are not intended, to ensure the order in which processes start up.)

In MACRO-11, a static process is defined by the Define Static Process (DFSPCS\$) assembly-time macro; see Section 3.11. This macro produces a block of information used by the memory image builder (MIB) utility and the kernel's INIT procedure. This information includes the initial address of the instruction sequence to be executed, the amount of stack space to be allocated, the runtime process name, mapping type, priority, and other characteristics specified in the macro call.

A MACRO-11 static process can implement the same kind of special, system-level initialization "procedure" as described above for Pascal, using the following strategy. The process starts up at priority 255, as specified in the DFSPCS\$ macro, in order to execute its initialization code. Immediately after the initialization processing, it uses the Change Priority (CHGPs\$) primitive to drop its priority to the desired operating level; it can now enter its main control loop, corresponding to the Pascal main program body.

A dynamic process is created by the action of another process during system execution. This action consists of a request to the kernel's process-creation service, which creates a process control block and schedules the new process. The kernel allows a static process to create one or more dynamic processes, each of which can in turn create other dynamic processes. The created process is essentially a subprocess of its creator, in the sense that the instruction and data segments of the created process must be located within the address space of the creating process. In a mapped environment, a dynamic process necessarily inherits the mapping type of its parent, or originating static process, since it shares the virtual address space of that static process. Thus, a static process can create a family of dynamic processes to handle a set of related asynchronous events; such processes may share common data areas.

In Pascal, each process-invocation statement is an implicit request for creation of a dynamic process. The process-invocation statement consists of a reference to an identifier defined by a PROCESS declaration, plus optional process attributes and invocation parameters. (Although a process invocation is syntactically similar to a procedure call, it initiates a control flow that is separate and distinct from that of the invoking process, as opposed to a transfer of control within the calling process. Flow of control cannot be

## PROCESSES AND SYSTEM DATA STRUCTURES

explicitly transferred from one process to another.) The PROCESS declaration defines the instruction sequence and local variables to be associated with a process created by a reference to that declaration. Multiple dynamic processes can be based on the same PROCESS declaration; separate instances of the local variables are allocated for each process.

In MACRO-11, a dynamic process is created by a Create Process (CRPCS) service request; see Section 3.7. This request specifies the initial address of the instruction sequence to be executed, the stack address, runtime process name, priority, and other characteristics of a dynamic process.

Static and dynamic processes are functionally equivalent; all kernel primitives are available to both kinds of processes. In particular, any process can delete itself. (This is the only valid way in which a process can terminate, assuming that termination is ever required.) The MicroPower/Pascal kernel does not enforce any hierarchical relationships between the members of a process family. Thus, any process can outlive its creator; no restrictions exist on the order in which related processes may terminate (if any of them must do so).

The MicroPower/Pascal compiler and object-time system does, however, impose its own structure on a process family with respect to the scope of identifiers and the longevity of process-local variables. The compiler applies the same nesting rules to PROCESS declarations as to PROCEDURE declarations. This allows the compiler to control the scope of variables accessed by processes at various levels, in a manner consistent with standard Pascal syntax rules. Furthermore, variables that are local to a dynamic process are allocated from dynamic storage -- the program's memory heap -- when the process is created, unless they are declared with the STATIC attribute. The storage for these variables is automatically released -- returned to the heap for reuse -- if and when the process terminates and is deleted.

Outer-level local variables, however, can be accessed by a nested process (one declared at an inner level). Therefore, the Pascal compiler/OTS ensures that an outer-level process will never be deleted -- will not actually terminate -- before its nested processes terminate, although it may have stopped executing. That is to say, the storage for a given process will not be released, and the process will not be deleted, until all lexically lower-level processes terminate, even though the process has logically terminated either by "reaching its END statement" or by returning from its termination procedure.

Note that variables declared at the outermost -- static process -- level remain available to any and all subprocesses until every member of the process family terminates.

### 2.1.2 Process Names

One process can refer to another in a limited number of kernel primitive requests -- for example, in a Suspend Process or Resume Process request. To facilitate such references, especially across process families, a process can be given a runtime name in the program that defines the process. A runtime process name consists of a 6-character ASCII string -- for example, 'ALPHA5' -- that is dynamically associated with the process when it is created. (The name is associated with the process control block corresponding to the process.) The string 'ALPHA5' can be used in primitive requests in another program to refer to the process globally known by that name.

## PROCESSES AND SYSTEM DATA STRUCTURES

Process names must be unique not only among all named processes throughout the system, but also among all named system structures. That is, a process name must not duplicate the name of any coexisting semaphore or ring buffer. Violation of this rule will cause errors during execution. (The names of system structures created by DIGITAL-supplied processes, such as device handlers, always contain a \$ character. Therefore, the \$ character should be avoided in all user-specified names.)

Since runtime names are fixed-length character strings, both case and trailing blanks are significant. Thus, the name 'abcl23' is not equivalent to 'ABC123', and 'ABCD ' is not equivalent to 'ABCD'.

In Pascal, a static process gets its runtime name from the compile-time program name specified in the program heading; this name is either truncated to six characters or, if less than six characters, padded with trailing spaces. A dynamic process gets its runtime name, if any, from a NAME attribute, specified in either a PROCESS declaration or a process-invocation statement. A name assigned at the point of process invocation overrides the default runtime name, if any, specified in the corresponding PROCESS declaration. See the MicroPower/Pascal Language Guide, Chapter 10, for a detailed description of the NAME attribute.

In MACRO-11, a runtime name is specified directly in the Define Static Process (DFSPCS\$) macro call and indirectly in the Create Process (CRPCS\$) service request. See Section 3.1.6 for a discussion of the process descriptor block.

### 2.1.3 Process States

Every process is in one and only one state at any given time. The seven process states supported by the kernel are:

1. Run: the state of the process currently in possession of the processor. This process may be executing at process level, executing a primitive operation, or may be interrupted. (An interrupt does not cause a transition from run state.) By definition, the priority of the running process is at least equal to that of any process in the ready-active state. The running process continues in the run state until it blocks or suspends itself or is preempted by a higher-priority process becoming ready to execute.
2. Ready active: the state of a process that is ready to execute and is eligible for the processor to be assigned to it. The highest-priority ready-active process is assigned the processor whenever the running process relinquishes control or is preemptable.
3. Ready suspended: the state of a process that is otherwise ready to execute but has been explicitly suspended by itself or by another process. A Resume operation by another process increments a suspend counter associated with the suspended process. When the suspend count changes from -1 to 0, the suspended process is returned to the ready-active state.
4. Wait active (blocked): the state of a process forced to wait -- defer execution -- until a particular event occurs or a given resource becomes available. When unblocked, the process changes to the ready-active state. See Section 2.1.4.2.

## PROCESSES AND SYSTEM DATA STRUCTURES

5. Wait suspended: the state of a process that was blocked -- forced to wait for an event or a resource -- and has subsequently been suspended by another process. A Resume operation by another process increments a suspend counter associated with the suspended process. When the suspend count changes from -1 to 0, the suspended process is returned to the wait-active state. If the process becomes unblocked while suspended, it changes to the ready-suspended state.
6. Exception wait active: the state of a process when an exception condition occurs that must be dispatched to an exception handler. The process must be removed from execution in order to allow the exception-handling process to execute. However, the process is not blocked in the strict sense of waiting on a semaphore or a ring buffer. The exception-wait state indicates that the process is waiting for action by an exception-handling process, as described in Section 2.1.4.4. The waiting process is placed in ready-active state when the exception handler "dismisses" the exception condition.
7. Exception wait suspended: the state of a process explicitly suspended while in the exception-wait-active state. A Resume operation increments a suspend counter associated with the suspended process. When the suspend count changes from -1 to 0, the suspended process is returned to the exception-wait-active state. If the exception handler dismisses the exception while the process is suspended, the process is placed in ready-suspended state.

When created, a process is in the ready-active state. The possible subsequent state transitions can be summarized as follows:

From	To
Ready active	Run (by priority) Ready suspended (by suspension)
Run	Ready active (by preemption) Wait active (by blocking) Exception wait active (by exception) Ready suspended (by self-suspension) Nonexistent (by deletion)
Wait active	Ready active (by unblocking) Wait suspended (by suspension)
Exception wait active	Ready active (by dismissal) Exception wait suspended (by suspension)
Ready suspended	Ready active (by resumption)
Wait suspended	Ready suspended (by unblocking) Wait active (by resumption)
Exception wait suspended	Ready suspended (by dismissal) Exception wait active (by resumption)

Figure 2-1 shows all state transitions and the events associated with them. The numbers indicate the kind of event, or the condition, that can cause the state transition represented by each arc. An asterisk preceding the number denotes a significant event, which causes the scheduler to be invoked.

## PROCESSES AND SYSTEM DATA STRUCTURES

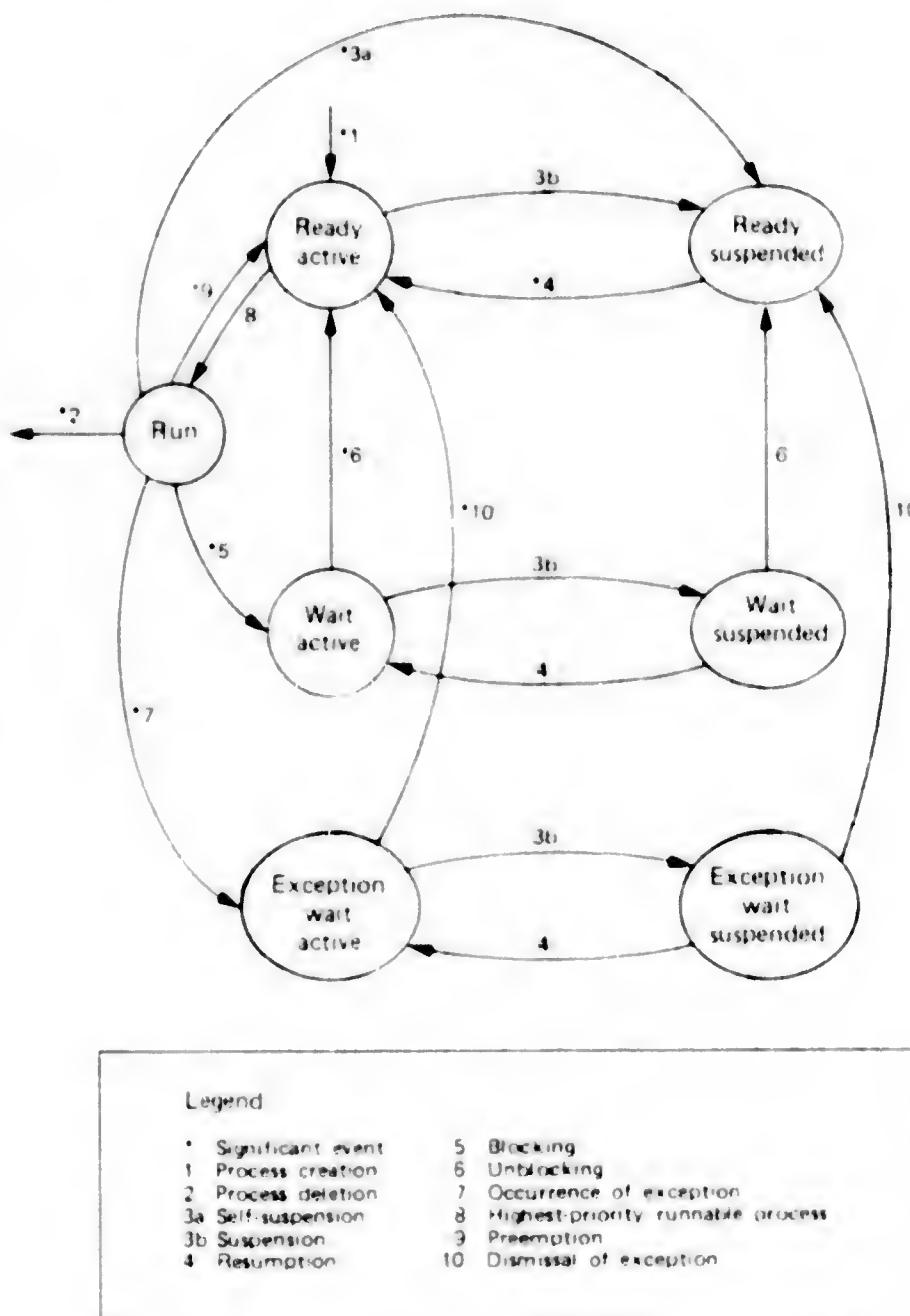


Figure 2-1 Process State Transitions

**2.1.3.1 Process State Codes** - The state of a process is described by the state code byte in its PCB (field PC.STA); see Section 2.1.5. The state code values are represented by the following global symbols:

State Code	Process State
SC.RUN	Run
SC.RDA	Ready active
SC.RDS	Ready suspended
SC.WTA	Wait active
SC.WTS	Wait suspended
SC.EWA	Exception wait active
SC.EWS	Exception wait suspended
SC.IAC	Inactive (aborted abnormally due to unhandled exception)

(These symbols are defined by the QUEDEF\$ system macro.) Whenever a process changes state, the kernel modifies the state code in its PCB. For most state changes, the kernel must also transfer the PCB from one state queue to another, as described below.

## PROCESSES AND SYSTEM DATA STRUCTURES

**2.1.3.2 State Queues** - The kernel maintains several queues -- linked lists -- of PCBs, called state queues. Each such queue reflects the state of the PCBs linked into it, although not every process state has a state queue. The PCB of every process that is not in exception-wait state is linked into one and only one state queue.

Conceptually, there are only four state queues, although the so-called wait queue consists of many distinct queues. The characteristics of these four queues are as follows:

1. Run queue: a degenerate, singly linked list that contains at most one element -- the PCB of the running process.
2. Ready-active queue: a doubly linked list of all ready-active PCBs, ordered according to process priority.
3. Ready-suspended queue: a doubly linked list of all ready-suspended PCBs, in LIFO order. (Note that the ordering of this queue is immaterial in that it has no bearing on the order in which processes may be resumed, that is, removed from the queue.)
4. Wait queue: a logical entity representing the collection of all waiting process lists associated with existing semaphores and ring buffers. Every semaphore has one waiting process list; the first word of a semaphore structure is the list header (see Sections 2.2.1.3 and 2.2.3). Every ring buffer has two waiting process lists, one for input and one for output, as described in Section 2.2.1.6.

A waiting process list, also called a blocking queue, is a singly linked list of all PCBs blocked on the associated structure. The PCBs on a given blocking queue may be queued in either FIFO or priority order, depending on the queuing characteristics specified for that structure. Also, the PCBs may be in either wait-active or wait-suspended state. Thus, the nominal wait queue comprises all waiting processes, whether active or suspended.

Note that there is no state queue for processes in an exception-wait state. The PCB of a process entering exception-wait-active state is passed to the appropriate exception-handler process and remains in the possession of that handler until it is returned to the ready state.

A uniform characteristic of MicroPower/Pascal priority-ordered queues is that queue elements -- for example, PCBs -- of equal priority are FIFO-ordered relative to each other. Thus, if several ready-active processes have the same priority value, the process with the "most time on queue" has the highest effective scheduling priority.

### 2.1.4 Process Scheduling

**2.1.4.1 Process Preemption** - The running process is preempted, or displaced from run state, if a higher-priority process becomes ready active. This can happen if either the running process or an interrupt service routine performs an operation that unblocks a wait-active process, resumes a ready-suspended process, or "dismisses" an exception-wait-active process.

## PROCESSES AND SYSTEM DATA STRUCTURES

Many kinds of semaphore and ring buffer operations -- a signal operation, for example -- can change a waiting process to ready active; the RESUME operation may, of course, change a suspended process to ready active. If the newly ready process is of higher priority than the running process, the former switches immediately to run state, and the latter reverts to the ready-active state. Preemption is always associated with the execution of certain primitive operations.

**2.1.4.2 Process Blocking and Unblocking** - A running process is said to block when it must give up the CPU in order to wait for a signal or a resource to be provided by another process. Thus, a process blocks for synchronization purposes; the blocking is always associated with execution of an unconditional, wait-type primitive operation on a semaphore or ring buffer. The kernel changes the process's state code from run to wait active and queues the PCB on the blocking queue of the semaphore or ring buffer.

The running process potentially allows itself to block by executing any of the primitive operations listed below. The MicroPower/Pascal predeclared procedure name for each operation is followed, in parentheses, by the corresponding MACRO-11 primitive request name.

- An unconditional wait operation on a binary or a counting semaphore:

WAIT procedure (WAITS request)

Blocking condition: The semaphore was not open -- not already signaled -- at the time of the wait operation.

- An unconditional Get Packet or Receive Data operation on a queue semaphore:

GET\_PACKET procedure (WAIQS request)  
RECEIVE procedure (RCVDS request)  
RECEIVE\_ACK procedure (RCVDS request)

Blocking condition: A packet was not available at the time of the Get or the Receive operation.

- An unconditional Get Element or Put Element operation on a ring buffer:

GET\_ELEMENT procedure (GELMS request)  
PUT\_ELEMENT procedure (PELM\$ request)

Blocking condition: Either too few buffer elements were available at the time of a Get Element operation, or too little buffer space was available at the time of a Put Element operation.

Note that the conditional forms of the operations listed above -- for example, COND\_WAIT function or WAICS request -- never cause the executing process to block.

A blocked process is unblocked by a primitive operation that provides the signal or resource for which the process was waiting. Unblocking implies a transition from either the wait-active or wait-suspended state to the corresponding ready state. The PCB of the unblocked process is moved to the appropriate ready state queue. As noted above, unblocking a wait-active process may in turn cause preemption

## PROCESSES AND SYSTEM DATA STRUCTURES

of the running process. The primitive operations that may unblock a waiting process are the following:

- A signal operation on a binary or a counting semaphore:

SIGNAL procedure (SGNLS request)  
COND\_SIGNAL function (SGLCS request)  
SIGNAL\_ALL procedure (SALLS request)

- A Put Packet or a Send Data operation on a queue semaphore:

PUT\_PACKET procedure (SGLQS request)  
COND\_PUT\_PACKET function (SGQCS request)  
SEND procedure (SENDS request)  
COND\_SEND function (SNDCS request)  
SEND\_ACK procedure (SEND\$ request)  
COND\_SEND\_ACK function (SNDCS request)

- A Get Element or a Put Element operation on a ring buffer:

GET\_ELEMENT procedure (GELMS request)  
COND\_GET\_ELEMENT function (GELCS request)  
PUT\_ELEMENT procedure (PELMS request)  
COND\_PUT\_ELEMENT function (PELCS request)

Unblocking conditions: A Get Element operation will unblock a process waiting to put elements into the same buffer if it frees enough space to satisfy the requirements of the Put operation. Conversely, and more obviously, a Put Element operation will unblock a process waiting to get elements from the same buffer if it supplies enough elements to satisfy the requirements of the Get operation.

Note that when successful, the conditional form of the semaphore operations listed above always unblocks a process, since the operation is performed only if a process is waiting on the semaphore.

**2.1.4.3 Process Suspension** - The running process can suspend itself or another process -- if the latter is currently active -- by requesting a SUSPEND (SPNDS) operation. In the case of self-suspension, the kernel changes the state code of the running process to ready suspended (SC.RDS) and moves its PCB to the ready-suspended queue. If the subject process was in the ready-active state, its PCB is similarly moved to the ready-suspended queue with the state code SC.RDS. If the subject process was either wait active or exception wait active, however, suspension involves only a modification of the state code to the suspended version of the previous state, with no movement of the PCB from one queue to another. The PCB of a waiting process remains on the same blocking queue throughout any transitions from active to suspended, or vice versa.

The Suspend and Resume operations modify the value of a suspend counter associated with each process. The value of the suspend counter is initially 0; a Suspend operation decrements this value, and the Resume operation increments it. An active process is in fact suspended only when its suspend count changes from 0 to -1, and a suspended process is in fact resumed only when its suspend count changes from -1 to 0. Therefore, a particular Suspend operation may not effectively suspend the subject process; conversely, a particular Resume operation may not effectively resume it, depending on the sequence in which preceding Suspend or Resume operations, if any, have been executed. (See the SPNDS and RSUM\$ primitives in Chapter 3.)

## PROCESSES AND SYSTEM DATA STRUCTURES

2.1.4.4 **Exception Handling** - The MicroPower/Pascal kernel defines certain processor traps as exception conditions that can be intercepted by an exception-handling process. The processor traps so defined are the following:

Vector	Name and Description
000	Vector fetch trap: SBC-11/21 only
004	Trap to 4: Bus timeout; for example, nonexistent memory address or invalid addressing mode
010	Trap to 10: Illegal and reserved instructions
014	BPT or T-bit instruction trap
024	Power-fail trap
030	EMT instruction trap
034	TRAP instruction trap
114	Memory parity error: LSI-11/23 only
140	Break trap to 140: SBC-11/21 only
244	Floating-point exception: FP-11 or FIS option
250	Memory-protection error: MMU option, LSI-11/23 only

In addition, a set of software exceptions are defined for processes implemented in MicroPower/Pascal. A complete list of exception types and subcodes is given in Table 7-1.

Further, the kernel permits a process to establish itself as an exception handler that services a particular type of exception condition for processes belonging to a given exception-handling group. (All processes have an exception-group attribute that is specified during process creation.) Exception handlers establish themselves through the use of either the Pascal CONNECT\_EXCEPTION procedure or the MACRO-11 Connect to Condition (CCND\$) request; the latter is described in Chapter 3.

Finally, assume that a running process of exception-handling group g causes an exception condition of type t to occur. That process will be placed in the exception-wait-active state only if an exception handler exists for exceptions of type t caused by a process of group g. If so, the PCB of the process is passed to the handler via its exception queue semaphore, for disposition according to the management strategy implemented by that handler. (See the DEXCS\$ request in Chapter 3 for further details.)

If no such handler exists, the faulting process remains in run state, but its flow of control is redirected by the kernel as follows:

1. The process is reentered at its exception service routine -- or Pascal exception service procedure -- if any. The process stack will contain information related to the exception condition.
2. If no exception service routine or procedure has been established, the process is reentered at its termination entry point, as if a Stop Process (STPC\$) request had been issued for the process.

An exception service routine is established for a process or for a family of processes by the Set Exception Routine Address (SERAS\$) primitive, as described in Chapter 3. For a process implemented in Pascal, an exception service procedure is established by the ESTABLISH predeclared procedure.

## PROCESSES AND SYSTEM DATA STRUCTURES

2.1.4.5 **Scheduler** - The scheduler is responsible for switching a ready-active process into the run state. The scheduler runs whenever a significant event -- one that could affect the ability of the running process to continue execution -- occurs in the system. There are three categories of significant events, as follows:

- A primitive executed by the running process that causes it to leave run state, switching to wait-active, ready-suspended, or ready-active state
- A primitive executed by either the running process or an interrupt service routine that causes another process to enter ready-active state
- Occurrence of an exception condition that is dispatched to an exception-handling process, causing the running process to enter exception-wait-active state

If the run queue is vacant when the scheduler executes, it moves the first -- highest-priority -- PCB from the ready-active queue to the run queue and restores the context of the new running process. Otherwise, it compares the priority of the PCB at the head of the ready-active queue with that of the PCB on the run queue to determine whether the running process should be preempted. If so, the scheduler makes the necessary queue change for both PCBs, placing the previously running process on the ready-active queue in proper priority order. It also performs a process-context switch, saving and restoring the context of the old and new running processes, so that the latter will gain control of the CPU on return from kernel processing.

### 2.1.5 Process Control Block (PCB)

A process is physically represented within the kernel by a process control block (PCB), the system data structure that identifies a particular activation of some instruction segment. The kernel creates a PCB in system-common memory when a process is created. The PCB is always linked into one of the kernel's state queues unless the process is in an exception-wait state, as previously described. The PCB serves a number of functions:

1. Defines the name, if any, and all other fixed attributes of the process.
2. Contains all dynamic state information maintained by the kernel concerning the process. This collection of information is called the software context of the process.
3. Provides a save area for kernel context that must be saved on a per process basis under certain circumstances.
4. Provides the save area for process context switching. The full hardware context of the process is saved in the PCB when the kernel switches the process out of run state. This context includes the contents of all registers that must be restored when the process is switched back to run state. (Note that the R3, R4, R5, PC, and PS values are saved in an interrupt stack frame on the process stack whenever the process is either interrupted or switched out of run state.)

Most primitive operations affect the content of a PCB, either directly, as in the case of process-management primitives, or indirectly, as when a primitive causes process blocking or unblocking.

## PROCESSES AND SYSTEM DATA STRUCTURES

Figure 2-2 shows how the PCB is organized.

Pointer -----> to PCB	+-----+   PC.FLK   State queue forward link word  -----    PC.BLK   State queue backward link word  -----    PC.STA   PC.PRI   State code/Priority  -----    PC.STS   PC.TYP   Status bits/Mapping type  -----    PC.PNT   Pointer to parent process  -----    PC.EXC   Exception entry point  -----    PC.MSK   Exception-type bit mask  -----    PC.SPT   Pointer to blocking structure  -----    PC.ALK   All-process list link word  -----    PC.SPC   Suspend counter  -----    PC.RLK   Kernel-resumption link word  -----    PC.GRP   PC.CXW   Exception group code/Context-switch   bits  -----    PC.TER   Termination entry point  -----    PC.MCX   Memory location to be switched  -----    PC.GOS   Stack-overflow guard word  -----    PC.GUS   Stack-underflow guard word  -----    PC.EPC   Saved PC after stack exception  -----    PC.EPS   Saved PS after stack exception  -----    PC.BPC   Saved initial PC prior to stack   exception  -----    PC.ESF   Saved exception stack frame pointer   for unhandled exception  -----    PC.KSP   Saved kernel stack pointer  -----    PC.KSV   Saved kernel-primitive context (3   words in unmapped systems; 5 words   in mapped systems)  -----    PC.USV   Saved user context (5 words in   unmapped systems; 10 words in mapped   systems)  -----    PC.MAP   Mapping-register values   (in mapped systems only; 16 words)  -----    PC.STK   Per process kernel stack   (in mapped systems only; 38 words)  -----    PC.FSV   Floating-point registers   (optional; 25 words) +-----+
-----------------------------	---

Figure 2-2 Process Control Block (PCB)

## PROCESSES AND SYSTEM DATA STRUCTURES

The PCB fields shown in Figure 2-2 are described below. The fields noted as dynamic reflect the state of the process and constitute its software context.

Field	Description
PC.FLK	Forward pointer to the next PCB in the current state queue; dynamic
PC.BLK	Backward pointer to the previous PCB in the state queue; dynamic (unused when PCB is linked into a blocking queue)
PC.PRI	Process priority value (range 0 to 255); set during process creation; modified by the CHGPS primitive (Chapter 3)
PC.STA	Process state code; dynamic
PC.TYP	Process mapping type code: PT.GEN for general, PT.SYS for privileged, PT.DRV for driver, or PT.DEV for device access; set during process creation
PC.STS	State code modifier bits; dynamic
PC.PNT	Pointer to the PCB of the parent process; 0 if a static process; set during process creation
PC.EXC	Address of process's exception service routine; set by the SERAS primitive (Chapter 3)
PC.MSK	Bit mask of exceptions that the process will accept; set by SERAS
PC.SPT	Pointer to semaphore or ring buffer that the process is blocked on, if any; the value in this field is valid only when the state code -- in field PC.STA -- is either SC.WTA or SC.WTS; dynamic
PC.ALK	Pointer to the next PCB in the list of all current processes
PC.SPC	Suspend count; modified by the SPNDS and RGUMC primitives (Chapter 3); dynamic
PC.RLK	Pointer to the next PCB in the kernel-resumption list (used by the kernel to queue processes awaiting "kernel resumption" following certain unblocking operations)
PC.GRP	Exception group code; set during process creation (see CRPCS or DFSPCS)
PC.CXW	Context-switch option bits; set during process creation (see CRPCS or DFSPCS primitive in Chapter 3)
PC.TER	Termination entry point; set during process creation
PC.MCX	Address of optional user-memory location to be saved in PC.USV; zero value if CXSMCX option was not selected; set during process creation
PC.COS	Lower boundary address for stack overflow checking
PC.GUS	Upper boundary address for stack underflow checking

## PROCESSES AND SYSTEM DATA STRUCTURES

PC.EPC	PC at context switch following a stack exception
PC.EPS	PS at context switch following a stack exception
PC.BPC	PC at context switch prior to a stack exception
PC.ESF	Pointer to the exception stack frame for a process that incurred a fatal -- unhandled -- exception and was aborted
PC.KSP	Saved primitive stack pointer; this value points into the process stack in an unmapped system or into PC.STK in a mapped system (see note below)
PC.KSV	Save area for kernel-primitive context: in an unmapped system, three words for R4, R3, and RW; in a mapped system, five words for kernel-mode R4, R3, RW, PAR 2, and PAR 3 (see note below)
PC.USV	Save area for user context switch: in an unmapped system, five words for user SP, RW, R1, R2, and the optional memory location; in a mapped system, ten words for user SP, RW-R5, PC, PS, and the optional memory location

The following two PCB fields are present only in mapped systems:

PC.MAP	Memory-mapping register (PAR and PDR) values for the process; 16 words; set during process creation; user's MMU registers are saved here only if the CX\$MMU option was specified, indicating that the process modifies its mapping; the registers are always restored from this area, however
PC.STK	Per process kernel stack; 38 words; in a mapped system, the kernel uses this area as its stack for primitive operations; in an unmapped system, the process stack is used instead

The following portion of the PCB is present only if the CX\$FPP option was selected for the process, indicating that it uses the FP-II floating-point processor:

PC.FSV	Save area for floating-point registers; 25 words for the LSI-11/23 FP-II floating-point option (KEF11-A) only
--------	---

### NOTE

The PC.KSP and PC.KSV context values are valid only while a process is blocked on a queue semaphore or a ring buffer or is waiting for packet allocation. The kernel uses these context values when it switches the process from wait to its subsequent ready state. In contrast, the user-context (PC.USV) values are valid whenever the process is not in the run state.

## PROCESSES AND SYSTEM DATA STRUCTURES

The size of a PCB varies both by hardware environment and by floating-point processor (FPP) usage, as follows:

- For a process in an unmapped system:

Without FPP context, 70 bytes  
With FPP context, 152 bytes

- For a process in a mapped system:

Without FPP context, 252 bytes  
With FPP context, 336 bytes

Note particularly that a PCB is prefixed by a structure header that is common to all typed structures, as described in Section 2.2.1. The header adds five words to the total amount of space allocated for a PCB. Also, if the PCB represents a named process -- is a named structure -- a 4-word structure name block is prefixed to the header, as described in Section 2.2.1.

### 2.1.6 Memory Partitioning and Process/Program Segmentation

The MicroPower/Pascal kernel uses a memory layout technique designed to work effectively in a hardware environment having a mixture of read-only (ROM) and read/write (RAM) storage. If a target system includes both ROM and RAM, enough ROM must be configured as low memory to contain all the kernel pure code. From that point on, ROM and RAM areas may be configured as desired. (The physical addresses implemented by the memory configuration need not be contiguous; there may be "holes" in memory both within and between the ROM and RAM areas. Also, ROM and RAM may be interspersed in physical memory. However, interspersed ROM and RAM within the address space of a static process is not supported.)

Correspondingly, the address space of a process family must be partitioned into two segments: low and high. The low segment contains the process pure-code and pure-data sections and will be located in ROM, if any. The high segment contains the impure-data sections and will be located in RAM. Thus, the two segments of each process family -- static process and any dynamic subprocesses -- will be located in two physically separate memory regions in a ROM/RAM environment, as shown in Figure 2-3. The kernel's address space is partitioned in the same manner. (For simplicity each process family is represented as a single process.)

## PROCESSES AND SYSTEM DATA STRUCTURES

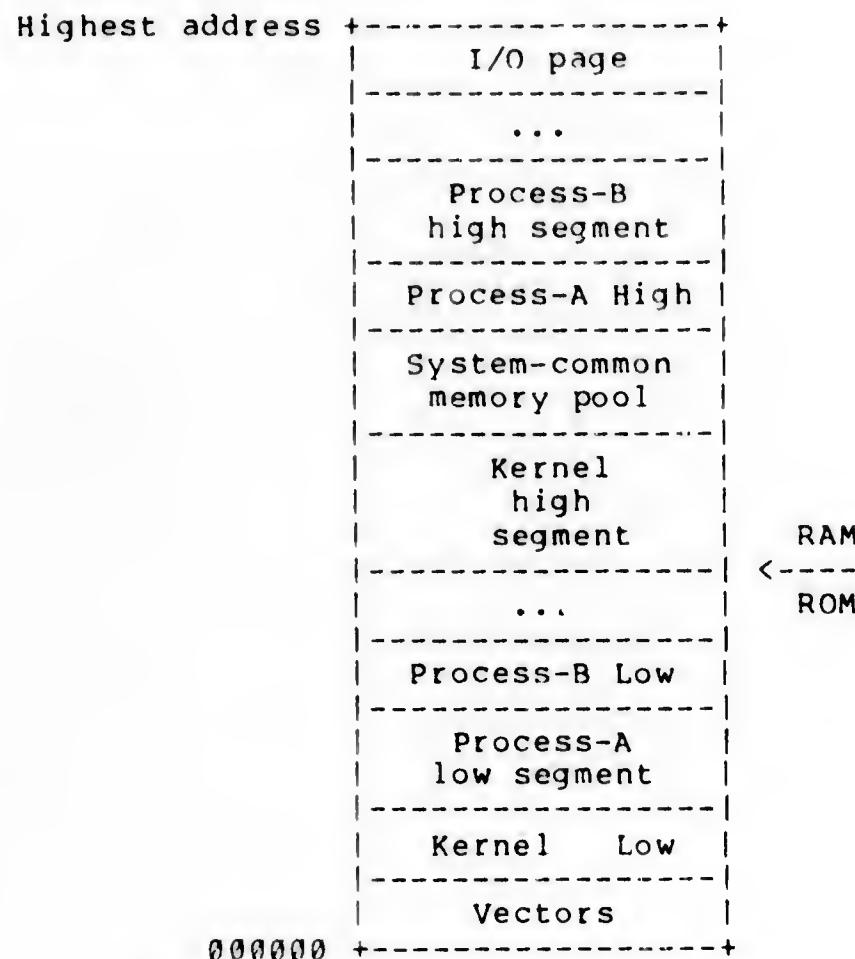


Figure 2-3 ROM/RAM Memory Layout

In a RAM-only system, the pure (low) and impure (high) segments are not physically separated in memory. The RAM-only memory layout is shown schematically in Figure 2-4.

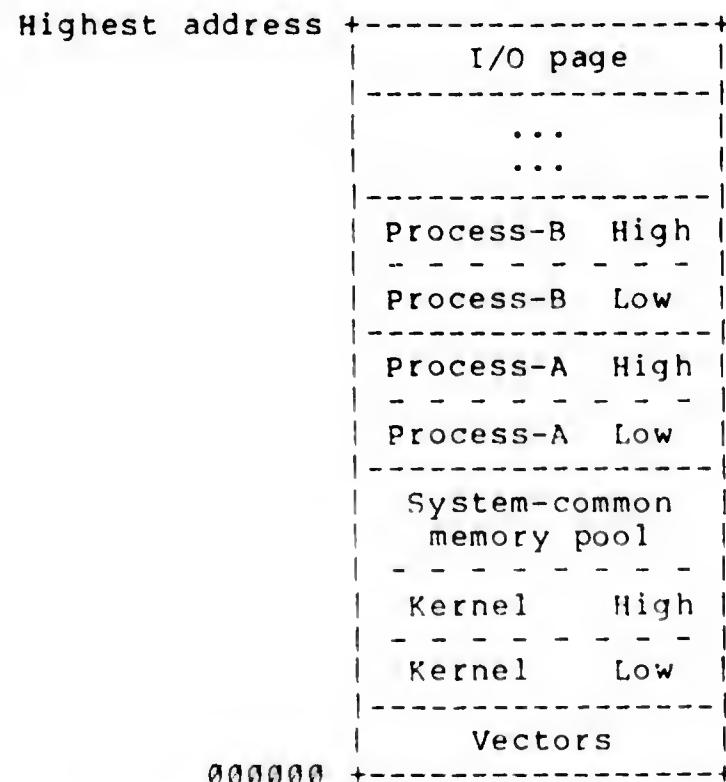


Figure 2-4 RAM-Only Memory Layout

## PROCESSES AND SYSTEM DATA STRUCTURES

To facilitate the low/high process code and data segmentation described above, an application program must segregate its code and data into appropriate read-only and read/write program sections, or p-sects. This must be done for any type of target-memory environment.

Program sectioning is provided transparently by the compiler for a MicroPower/Pascal program. The MACRO-11 programmer can use the PURE\$, PDAT\$, and IMPUR\$ assembly-time macros, preceding code, pure-data, and impure-data sequences, respectively, to conveniently generate the proper program sectioning directives. (See Chapter 3.) During the system build process, the RELOC utility groups the p-sects according to the read-only versus read/write attribute. (Within each group, p-sects are sorted into alphabetical order.)

In an unmapped environment, all processes have direct access to kernel memory space and in particular to the kernel's impure segment. The total system is normally limited to 56K bytes. (With an MVS11-DD or MVS11-ED memory module with the "extra 2K words" option enabled, 60K bytes of usable memory are available. This option halves the size of the I/O page.)

In a mapped environment, the MMU address-relocation hardware assists in memory segmentation and also provides memory protection. A mapped system with 18-bit physical addressing can support up to 252K bytes of usable memory, and a single process or process family can occupy up to 64K bytes. (A mapped system with 22-bit addressing can support up to approximately 4M bytes of memory.)

### 2.1.7 Process Mapping Types

The information in this section applies only to a mapped-memory hardware environment (for example, an LSI-11/23 target system).

"Mapping type" refers to the pattern of virtual-to-physical address translation used for a particular mappable object in the system. The mappable objects are the kernel, interrupt service routines (ISRs), and four types of processes: general, device access, driver, and privileged. More specifically, a mapping type identifies a particular page address register (PAR) usage convention associated with one of these objects. Both kernel mapping and ISR mapping use the kernel mode set of PARs. The four types of process mappings use the user mode set of PARs. (Figures 2-5 to 2-9 show the PAR assignments for each mapping type; these figures are discussed later.)

A mapping type is specified when a static process is defined. Any subprocesses created by the static process inherit its mapping type, since all the code and data associated with a given process family must reside in the same address space. Thus, from the viewpoint of process mapping, all dynamic processes are part of a parent static process; one set of address-relocation values is used for all processes within a family. The basic characteristics of the general, device-access, driver, and privileged process mappings are as follows:

1. General: the standard mapping for most application processes. General-process mapping is intended for processes that do not require direct access to system data structures or to the I/O page. General-process mapping allows for the largest possible static process or process family; the full range of virtual addresses -- 0 to 177777(octal) -- is available for process code and data. Therefore, the pure and impure segments defined for a static process and its subprocesses, if any, can total up to 64K bytes.

## PROCESSES AND SYSTEM DATA STRUCTURES

Due to hardware constraints, however, the high, or impure, segment in a mapped ROM/RAM target environment must begin on a 4K-word virtual address boundary. The restriction is enforced by the build utilities. Thus, a sizable "hole" in the virtual address space -- up to 8K bytes, less one byte -- may exist between the highest address in the low segment and the beginning of the high segment. (This is true for processes of any mapping type.)

In Pascal, if no mapping attribute -- DEV ACCESS, DRIVER, or PRIVILEGED -- is specified, the process family defined by the program has general mapping.

2. Device access: intended for processes that require access to the I/O page -- to device CSRs, MMU registers, and so forth -- but not to system data structures. Device-access mapping is suitable for a process that communicates directly with a dedicated I/O device, for limited device handling, or for a process that must modify its own mapping. Device-access mapping differs from general-process mapping only in that virtual addresses 160000 to 177777(octal) are mapped to the I/O page. This removes a 4K-word segment from the address space available for process code and data. Thus, the maximum size of a device-access process family is 56K bytes.

In Pascal, if the DEV ACCESS mapping attribute is specified, the process family defined by the program has device-access mapping.

3. Driver: intended for device-handling processes. Driver mapping allows direct access to system data structures -- to the kernel's common data space -- as well as to the I/O page. It also allows PARS 0 and 1 to be used as "scratch" address registers, for mapping to another process's input or output data-buffer area, for example. Driver mapping restricts process size to a maximum of 16K bytes but allows very efficient queue semaphore operations -- for interprocess message transmission -- and is compatible with the kernel-mode mapping of an interrupt service routine.

The lowest 16K bytes of virtual address space -- addresses 0 to 037777(octal) are available for any use -- for example, for mapping to a requesting process's buffer space. Virtual addresses 040000 to 077777 (16K bytes) are available for driver-process/ISR code and data. Virtual addresses 100000 to 157777 are mapped to the kernel's common data area (24K bytes), and addresses 160000 to 177777 are mapped to the I/O page (8K bytes).

In Pascal, if the DRIVER mapping attribute is specified, the process family defined by the program has driver mapping.

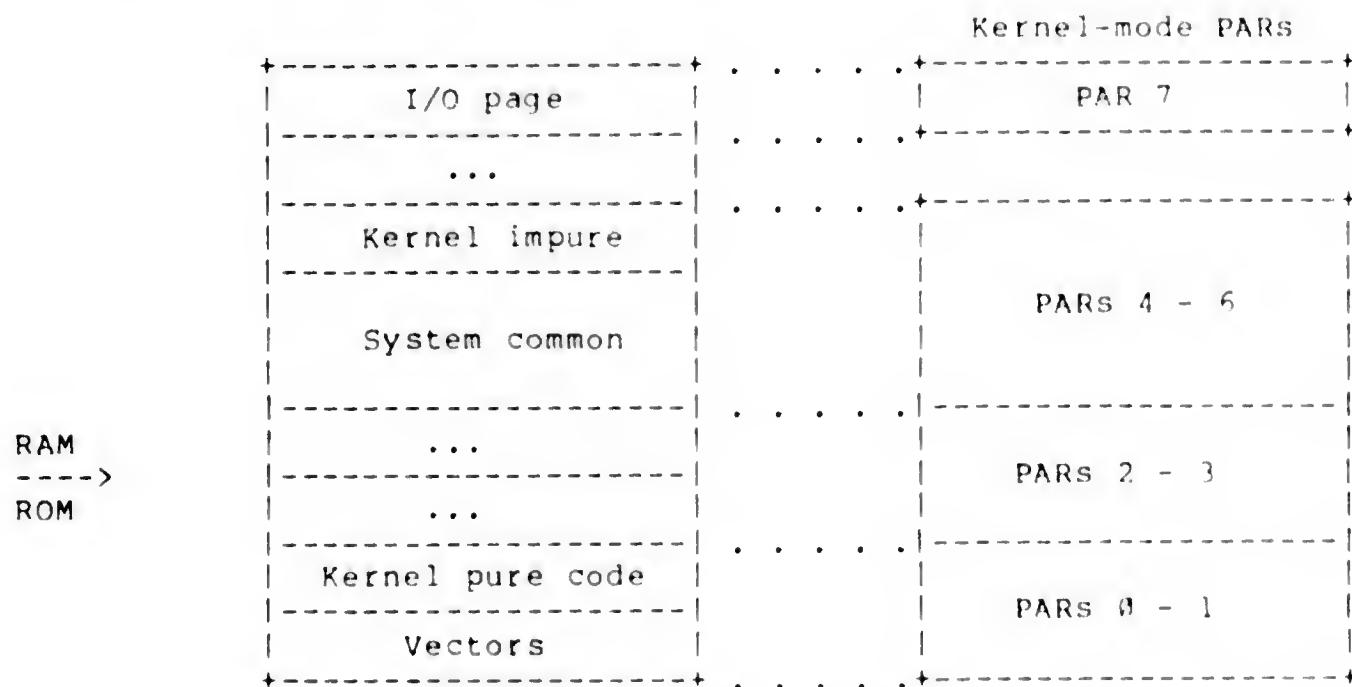
4. Privileged: intended for processes that need direct access to system data structures -- to the kernel's common data space -- as well as to the I/O page. Also called full-system mapping, privileged mapping restricts process size to a maximum of 32K bytes but allows very efficient queue semaphore operations, for interprocess message transmission. Privileged mapping is commonly used by processes that provide systemwide services other than device handling. This mapping is also used by exception-handling processes, which generally require access to PCBs.

## PROCESSES AND SYSTEM DATA STRUCTURES

The lowest 32K bytes of virtual address space -- addresses 0 to 077777(octal) -- are available for process code and data. Virtual addresses 100000 to 157777 are mapped to the kernel's common data area (24K bytes), and addresses 160000 to 177777 are mapped to the I/O page (8K bytes).

In Pascal, if the PRIVILEGED mapping attribute is specified, the process family defined by the program has privileged mapping.

Figure 2-5 illustrates kernel mapping. Kernel-mode PARs 0 and 1 map the hardware vector area and the kernel pure code segment; thus, the latter is limited to less than 8K words. PARs 2 and 3 are scratch address registers; they are modified as needed to map to user address space, for mapping user argument blocks, for example. PARs 4 to 6 map system-common memory and the kernel's own impure data, allowing 12K words of system data. (The system-common area consists of two memory pools in which the kernel allocates space for system data structures -- semaphores, ring buffers, PCBs -- and for queue packets.) Note that privileged and driver processes cannot access the kernel's impure data, although they can access the system-common area mapped by PARs 4 to 6. The kernel's own data is invisible to all other mapped objects. PAR 7 maps the I/O page, although the kernel does not access I/O devices. However, the memory-management registers reside in the I/O page, and the kernel must have access to them.

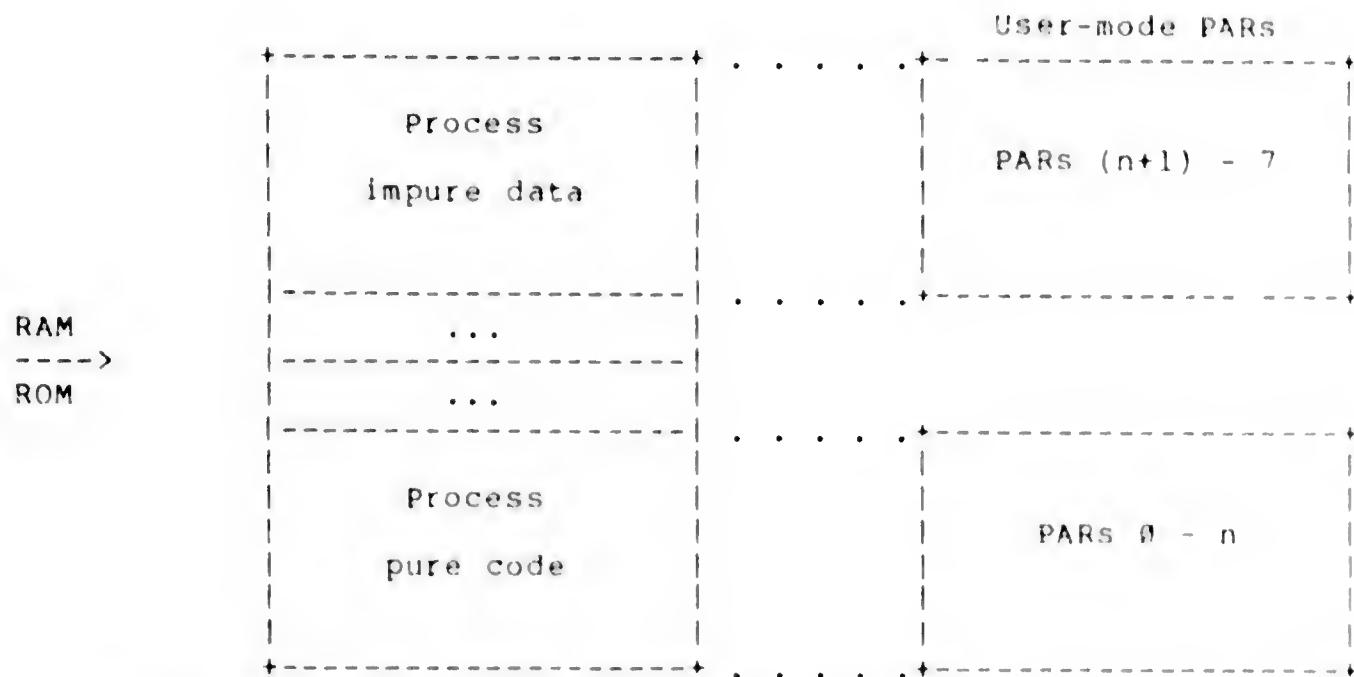


Kernel context: PC, PS, SP, R3-R5, kernel PARs 2 and 3.

Figure 2-5 Kernel Mapping

Figure 2-6 illustrates general-process mapping, which provides access only to user-defined memory. All eight user-mode PARs are available for mapping process code and data, allowing a maximum of 32K words of process space. Process pure code is mapped by PARs 0 to n, where n<7, allowing 4K\*(n+1) words of code (up to 28K). Process data is then mapped by the remaining PARs, (n+1) through 7, permitting 4K\*(7-n) words of data.

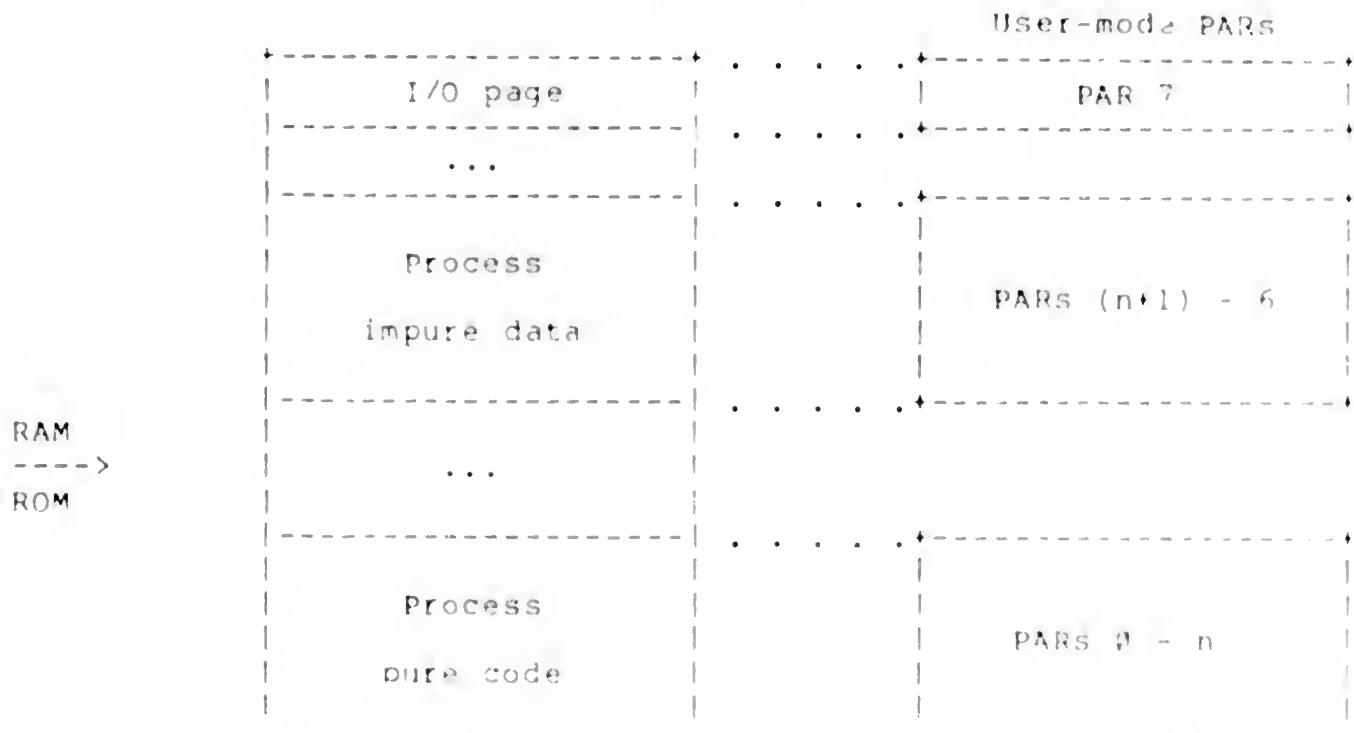
## PROCESSES AND SYSTEM DATA STRUCTURES



Process context: PC, PS, SP, R0-R5, user PARS 0 to 7.

Figure 2-6 General-Process Mapping

Figure 2-7 illustrates device-access process mapping, which provides access to the I/O page but not to the system-common area. User-mode PARS 0 to 6 are available for mapping process code and data, allowing a maximum of 28K words of process space. Process pure code is mapped by PARS 0 to n, where  $n \leq 6$ , allowing  $4K \times (n+1)$  words of code (up to 24K). Process data is then mapped by the remaining PARS, (n+1) to 6, permitting  $4K \times (6-n)$  words of data. PAR 7 maps the I/O page.

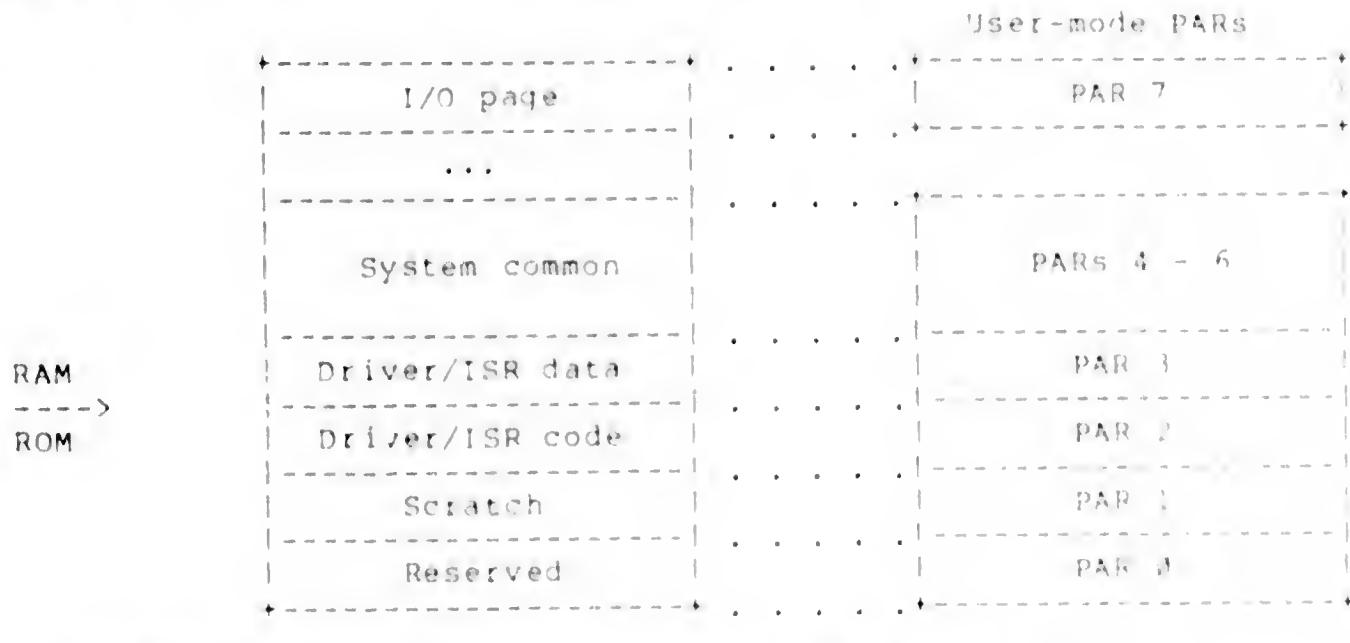


Process context: PC, PS, SP, R0-R5, user PARS 0 to 7.

Figure 2-7 Device-Access Process Mapping

## PROCESSES AND SYSTEM DATA STRUCTURES

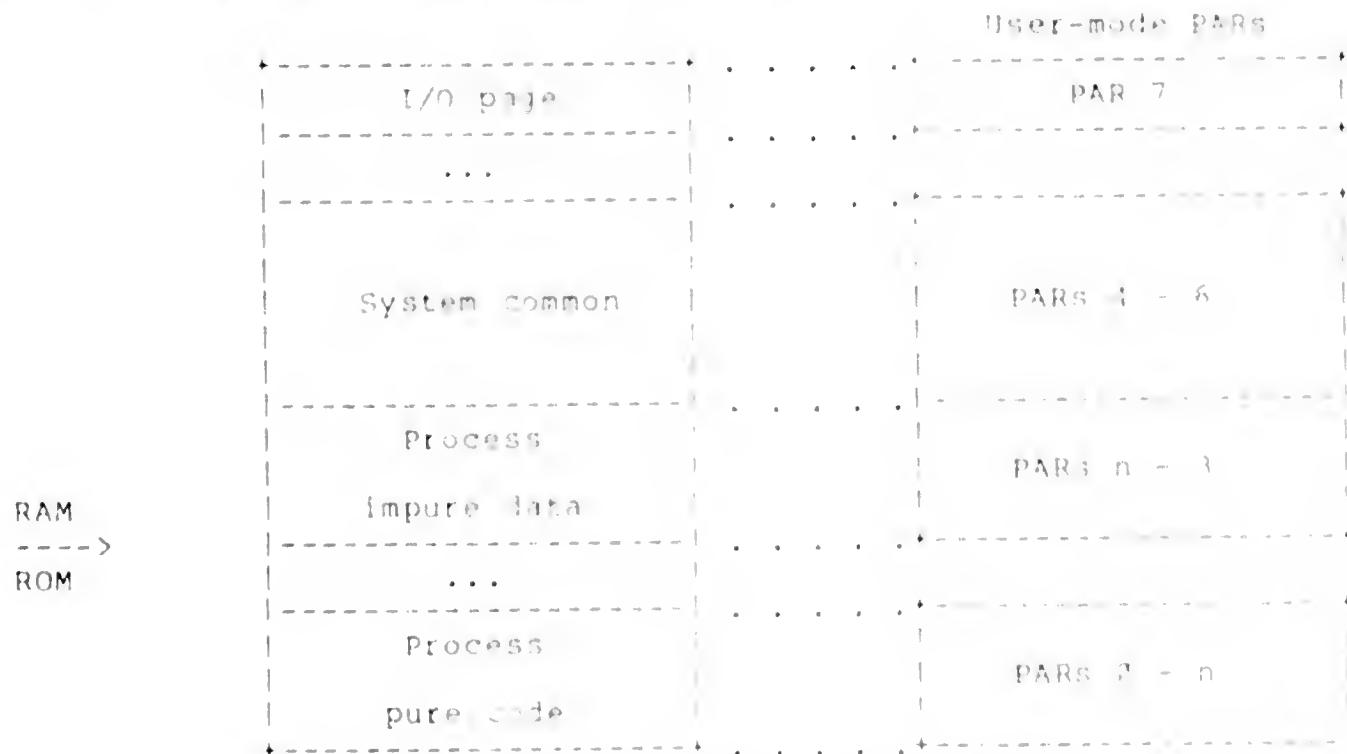
**Figure 2-8** illustrates driver process mapping, which provides access to both the I/O page and the system-common area. User-mode PAR 1 is a scratch register. PARS 2 and 3 map the process code and data, respectively, which may total 8K words. PARS 4 to 6 map the system-common data area. PAR 7 maps the I/O page. User-mode PAR 8 is reserved by DIGITAL for future device handler interfaces.



Process context: PC, PS, SP, R0-R5, user PARS 0 to 7.

**Figure 2-8** Driver Process Mapping

**Figure 2-9** illustrates privileged-process mapping, which provides access to both the I/O page and the system-common area. User-mode PARS 0 to 3 map the process code and data, which may total 16K words. PARS 4 to 6 map the system-common data area. PAR 7 maps the I/O page.

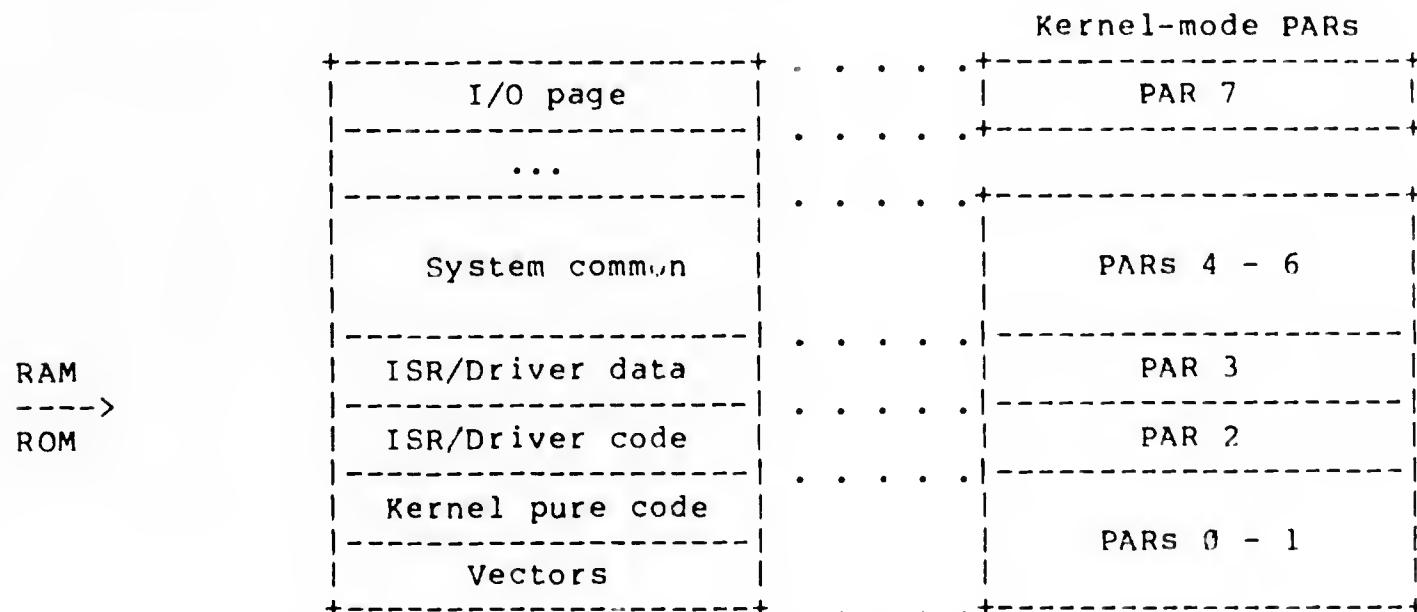


Process context: PC, PS, SP, R0-R5, user PARS 0 to 7.

**Figure 2-9** Privileged-Process Mapping

## PROCESSES AND SYSTEM DATA STRUCTURES

Figure 2-10 illustrates interrupt service routine (ISR) mapping; ISRs are described in Chapter 8. The mapping of ISRs uses the kernel-mode PARs and is designed to be very fast. Kernel-mode PARs 2 and 3 are saved and then set up to map the driver/ISR code and data. The rest of the mapping context remains that of the kernel; thus, the ISR is mapped to system common, the I/O page, and the kernel. PAR 1, which is mapped to kernel code, is available to the ISR -- can be borrowed -- for mapping to user buffers, but the ISR must save and restore it if so used. In particular, PAR 1 must, if borrowed, be restored before issuing a FORKS request.



ISR context: PC, PS, SP, R3-R5, kernel PARs 2 and 3.

Figure 2-10 Interrupt Service Routine Mapping

### 2.2 SYSTEM DATA STRUCTURES

The MicroPower/Pascal runtime system uses a variety of dynamic data structures. Instances of these structures are allocated by the kernel in system-common memory as a direct or indirect result of requests for kernel services. This section describes the format of these structures. However, you do not need to know how these structures are implemented in order to use the kernel services; the primitive-request interface hides this level of detail. This information is provided because it is often useful, and sometimes necessary, when debugging an application. In addition, you need some knowledge of kernel internals for designing and coding privileged system-level processes such as exception handlers.

The structures described here comprise the typed data structures -- for example, semaphores and ring buffers -- message packets, and the several kinds of queues -- linked lists of structures -- used by the kernel. The descriptions include the MACRO-11 symbolic offset names assigned to each element of a structure. The overall organization of the system-common memory area is also described.

Several kernel structures related to interrupt dispatching and exception dispatching are described in Chapters 7 and 8.

## PROCESSES AND SYSTEM DATA STRUCTURES

### 2.2.1 Typed Data Structures

The system data structures created and deleted by processes, via primitive operations, are called typed data structures. Each instance of a typed structure carries a structure-type code, used for validity checking, in its header. Six structure types are defined:

- Binary semaphore
- Counting semaphore
- Queue semaphore
- Ring buffer
- Process control block (PCB)
- Unformatted structure

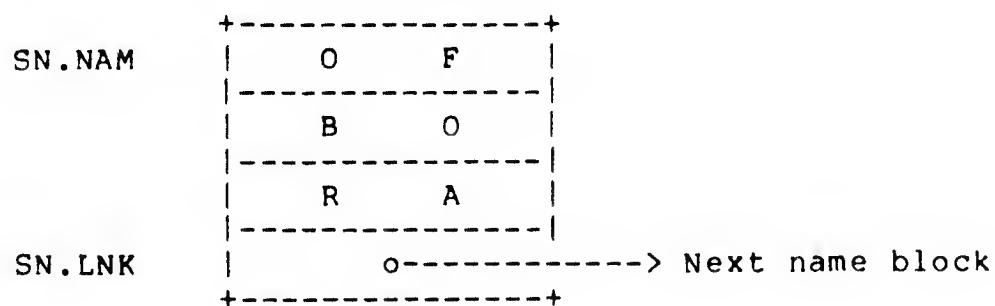
Except for PCBs, these structures are explicitly created and deleted by the Create Structure (CRSTS\$) and the Delete Structure (DLSTS\$) primitives. PCBs are implicitly created and deleted as a part of process creation and deletion. (The PCB was defined earlier in this chapter under the general discussion of processes.)

Other than creation and deletion, no kernel operations are defined on an unformatted structure. This type of structure is available for application-defined purposes.

All typed structures can be named. (The naming of PCBs is discussed in Section 2.1.2.)

**2.2.1.1 Structure Names and Name Blocks** - The kernel allows a runtime name composed of six ASCII characters to be dynamically associated with a typed structure when the typed structure is created. The name must be unique across all structures -- semaphores, ring buffers, and PCBs -- to which a runtime name is assigned. Once a named structure is created, any process in the system can refer to it by name when requesting operations on it. Such names facilitate source-time references to a given structure in several different application programs, which in a mapped environment represent processes in separate address spaces. In Pascal, a structure name can be specified directly in a structure-creation request and used in other requests for operations on the structure. Section 3.1.5 describes the use of structure names and the structure descriptor block in MACRO-11 programs.

Every named structure is prefixed by a 4-word structure name block. This name block precedes the standard structure header described in Section 2.2.1.2. The name block contents are set during structure creation. The format of the structure name block is as follows (FOOBAR represents a structure name):



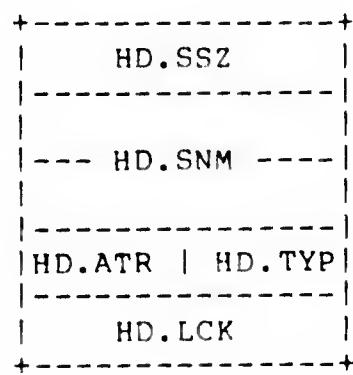
## PROCESSES AND SYSTEM DATA STRUCTURES

In the format above:

- SN.NAM contains the 6-character ASCII structure name.
- SN.LNK is a structure name table (SNT) link word.

The SN.NAM and SN.LNK symbols are defined by the QUESDF\$ system macro as negative offsets from the start of the structure body. (Runtime pointers to typed structures point to the actual structure body.) The symbol SN.SIZ defines the size of a structure name block in bytes. The symbol SN.CHR defines the number of characters in the SN.NAM field.

**2.2.1.2 Structure Header** - All typed structures have a standard prefix, or structure header. The header contents are set during structure creation and are never modified except for the HD.LCK field when implemented. The format of the structure header is as follows:



In the format above:

- HD.SS2 is the structure size, in bytes, including the header and is used during structure deallocation.
- HD.SNM is the structure serial number (32 bits), a value that is unique to each instance of a typed structure, and is used during structure name lookups for validity checking.
- HD.TYP is the structure-type code (defined below) and is used by various primitives for validity checking.
- HD.ATTR is the structure attribute bits (defined below).
- HD.LCK is reserved for future use.

The HD.xxx symbols are defined by the QUESDF\$ system macro as negative offsets from the start of the structure body. (Runtime pointers to typed structures point to the actual structure body, not to the header.) The symbol HD.SIZ defines the size of a structure header, in bytes.

## PROCESSES AND SYSTEM DATA STRUCTURES

The structure-type code (in HD.TYP) has the following range of symbolic values:

Code	Value
ST.BSM	Binary semaphore
ST.CSM	Counting semaphore
ST.QSM	Queue semaphore
ST.RBF	Ring buffer
ST.PCB	Process control block
ST.UDF	Unformatted structure

The structure-attribute bits (in HD.ATR) are defined as follows:

Code	Definition
SA\$NAM	Structure is named if set, unnamed if not
SA\$RIA	Ring buffer input access mode is stream if set, record if not; affects only Conditional Put Element (PELCS) operations
SA\$ROA	Ring buffer output access mode is stream if set, record if not; affects only Conditional Get Element (GELCS) operations
SA\$QUO	For type ST.QSM, packet-queue ordering is by priority if set and FIFO if not; for type ST.RBF, waiting-input-process list ordering is by priority if set and FIFO if not
SA\$PRO	For types ST.BSM, ST.CSM, and ST.QSM, waiting-input-process list ordering is by priority if set and FIFO if not; for type ST.RBF, waiting-output-process list ordering is by priority if set and FIFO if not

**2.2.1.3 Binary Semaphore Definition** - A binary semaphore consists of a binary variable and a singly linked list of waiting processes. Two operations on the variable are defined: signal and wait. The signal operation increments the semaphore variable. (The variable cannot assume a value greater than 1, however.) The wait operation decrements the semaphore variable if possible. If the value of the variable is 0, it cannot be decremented; binary variables can assume only the values 0 and 1. The process invoking this operation then waits until it is possible.

The format of a binary semaphore, excluding the structure header, is as follows:



In the format above:

- BS.FPT is the forward pointer to the first waiting process, if any.
- BS.VAR is the semaphore gate variable.

## PROCESSES AND SYSTEM DATA STRUCTURES

The SA\$PRO bit of the structure-header attribute byte (HD.ATR) must be set if waiting processes are to be queued in priority order.

**2.2.1.4 Counting Semaphore Definition** - A counting semaphore consists of a nonnegative integer variable, or counter, and a singly linked list of waiting processes. Two operations on the variable are defined: signal and wait. The signal operation increments the semaphore variable. The wait operation decrements the semaphore variable, if possible. If the variable is 0, it cannot be decremented; nonnegative variables cannot, by definition, assume values less than 0. The process invoking the operation must then wait until it is possible to decrement the variable. The counting semaphore differs from the binary semaphore only in that the semaphore variable can assume values greater than 1. Thus, n successive signal operations will allow n subsequent wait operations to proceed without waiting.

The format of a counting semaphore, excluding the structure header, is as follows:



In the format above:

- CS.FPT is the forward pointer to the first waiting process, if any.
- CS.CNT is the counter variable.

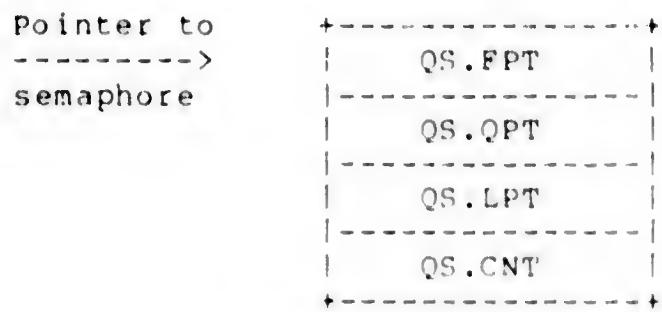
The SA\$PRO bit of the structure-header attribute byte (HD.ATR) must be set if waiting processes are to be queued in priority order.

**2.2.1.5 Queue Semaphore Definition** - A queue semaphore is a further generalization of a counting semaphore. In this case, there are two singly linked lists, one of waiting processes and another of available elements, or message packets. The two basic operations defined on queue semaphores are Put Packet and Get Packet. The Get Packet operation tests the element queue for an available element. If one is available, it is dequeued and passed to the requesting process. If no elements are on the queue, the process is blocked on the semaphore's waiting process list until one becomes available.

The Put Packet operation places an element on the semaphore's element queue. It first tests to see if a process is waiting. If so, it unblocks the process, moving it to the appropriate ready state queue, and passes the element pointer to the unblocked process. If no process is waiting for an element, the element is placed on the semaphore's element queue. The standard queue element, or message packet, is defined in Section 2.2.2.

## PROCESSES AND SYSTEM DATA STRUCTURES

The format of a queue semaphore, excluding the structure header, is as follows:



In the format above:

- QS.FPT is the forward pointer to the first waiting process, if any.
- QS.QPT is the pointer to the first element on the queue, if any.
- QS.LPT is the pointer to the last element on the queue, if any.
- QS.CNT is the count of the available queue elements.

The SASQUO bit of the structure-header attribute byte (HD.ATR) must be set if queue element ordering is to be by priority rather than by FIFO. The SA\$PRO bit of the attribute byte must be set if waiting processes are to be queued in priority order.

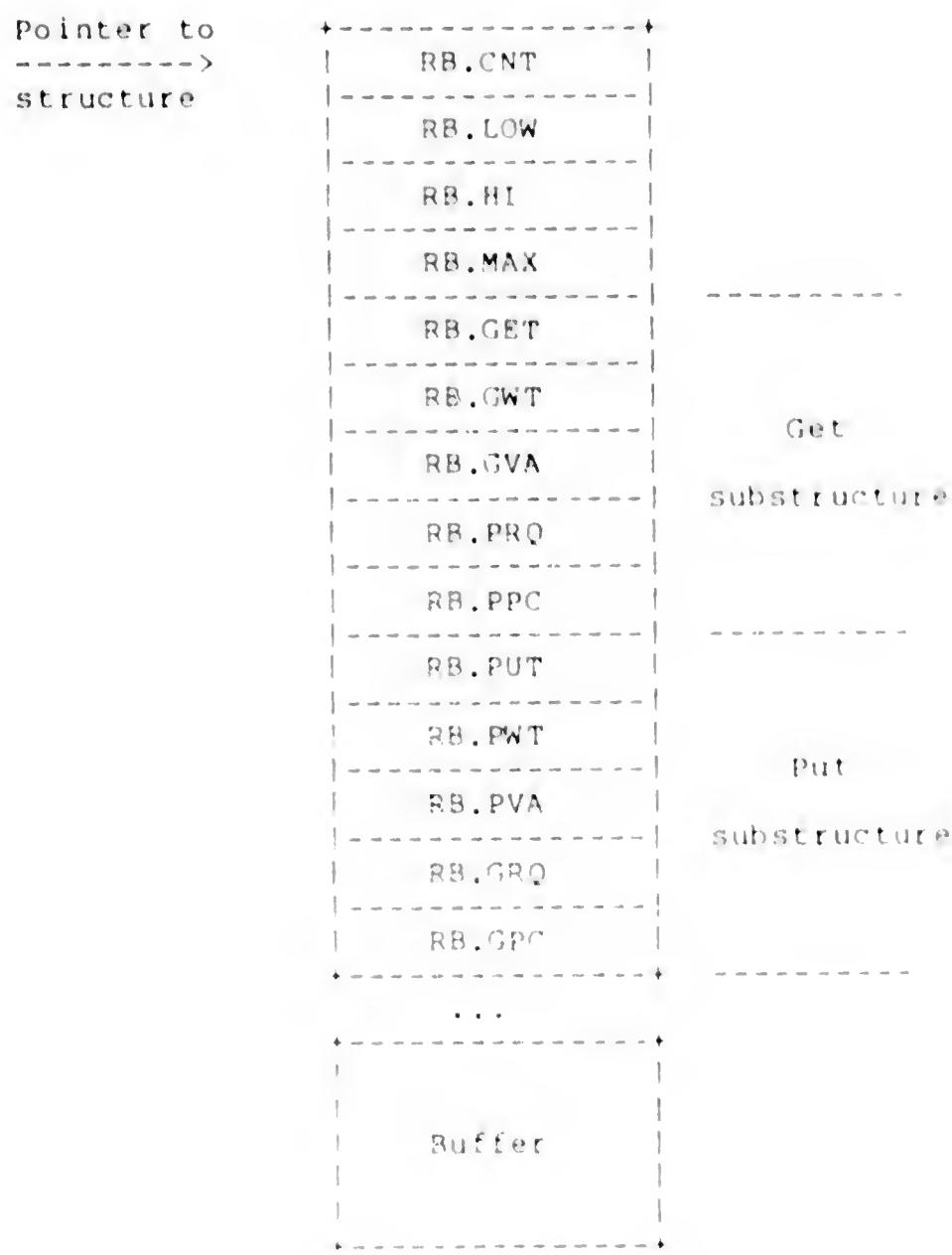
**2.2.1.6 Ring Buffer Definition** - A ring buffer consists of a control structure and a data buffer of user-specified size. The control structure includes a Get substructure that controls buffer output (Get Element) operations and a Put substructure that controls buffer input (Put Element) operations. The Get substructure has a waiting output-process list, or Get queue, and the Put substructure has a waiting input-process list, or Put queue.

The buffer, which is circular in the implementation sense, can be thought of as having both an input and an output end, such that two buffer-transfer operations can be in progress at the same time. For example, a process can be blocked on the output end of the buffer, waiting for sufficient data to satisfy its Get request, while another process is putting some number of bytes into the buffer at the input end. The reverse situation can also occur, of course, as when an input process must wait for space to become available. Once a process gains active access to the buffer, its input or output operation must complete before another process is given access to the same end of the buffer. If necessary, the requesting process will block on the buffer until the transfer is completed. Other processes attempting to access the same end of the buffer will be blocked behind the process whose transfer is in progress, regardless of their priority.

See the GELMS, GELCS, PELMS, PELCS, and RBUFS primitives in Chapter 3 for a complete description of ring buffer operations.

## PROCESSES AND SYSTEM DATA STRUCTURES

The format of a ring buffer, excluding the structure header, is as follows:



In the format above:

- RB.CNT is the count of bytes of data available for output.
- RB.LOW is the low limit -- the starting address of the buffer.
- RB.HI is the high limit -- the highest address of the buffer -- used to determine when to wrap the PUT or GET pointer around to the beginning of the buffer, creating the circular buffer structure.
- RB.MAX is the size of the buffer, that is, the maximum number of bytes.
- RB.GET is the pointer to the next available byte; used when removing an element from the buffer.
- RB.GWT is the pointer to the next process waiting for output access, if any (see RB.GPC).

## PROCESSES AND SYSTEM DATA STRUCTURES

- RB.GVA is the binary gate variable for Get operations.
- RB.PRQ is the variable used during concurrent Get/Put operations.
- RB.PPC is the pointer to the blocked process with current input access to the buffer, if any (the effective head of the Put queue).
- RB.PUT is the pointer to the next free location in the buffer; used when inserting elements into the buffer.
- RB.PWT is the pointer to the next process waiting for input access, if any (see RB.PPC).
- RB.PVA is the binary gate variable for Put operations.
- RB.GRQ is the variable used during concurrent Put/Get operations.
- RB.GPC is the pointer to the blocked process with current output access to the buffer, if any (the effective head of the Get queue).

The control structure and the buffer area may not be contiguous in memory.

The SA\$QUO bit of the structure-header attribute byte (HD.ATR) must be set if waiting input processes are to be queued in priority order rather than FIFO. The SA\$PRO bit of the attribute byte must be set if waiting output processes are to be queued in priority order. The SA\$RIA and SA\$ROA bit settings affect only the Conditional Put (PELCS) and Conditional Get (GELCS) operations, respectively, as described in Chapter 3.

**2.2.1.7 Process Control Block Definition** - The structure of a PCB is defined in Section 2.1.5.

**2.2.1.8 Unformatted Structure Definition** - An unformatted structure consists of a data area of user-specified size, preceded by a standard structure header. The kernel does not impose a format on the data area, as no primitives are provided to operate on it. An unformatted structure is allocated by the CRSTS primitive from system-common memory, has the unformatted type code, and may be named. Such a structure may be operated on directly by a privileged or a driver process.

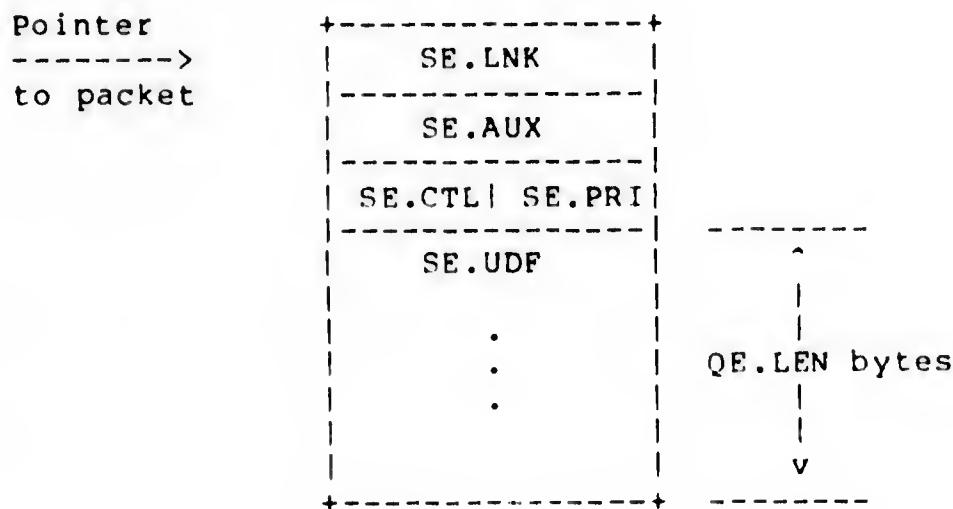
### 2.2.2 Message Packets

Standard, fixed-length queue elements, called packets, are used with queue semaphores to implement message transmission within the system. The kernel maintains a pool of free packets in the system-common area. A process obtains a packet from this pool by performing an ALLOCATE\_PACKET (or ALPK\$) primitive operation. When no longer needed, the packet must be returned to the free-packet pool by a DEALLOCATE\_PACKET (or DAPK\$) operation. Thus, a packet is a reusable (serially sharable) kernel resource.

## PROCESSES AND SYSTEM DATA STRUCTURES

A packet consists of a 3-word packet header and a fixed amount of message space called the undefined portion. Packet size is determined by a kernel assembly-time parameter, which can be modified by the user installation if a larger packet size is required. The size of a packet for an as-released system is 40 bytes; this allows 34 bytes of usable message space (undefined portion).

The format of a packet is as follows:



In the format above:

- SE.LNK is the forward link word for packet queuing (that is, the pointer to the next packet in a queue); set by the Signal Queue Semaphore (SGLQ\$) and Send Data (SEND\$) primitives.
- SE.AUX is the auxiliary link word; reserved for future use.
- SE.PRI is the packet priority value, if any; used by the SGLQ\$ and SEND\$ primitives if the packet queue is priority-ordered.
- SE.CTL is the message-format control byte; set by the SEND\$ primitive and used by the Receive Data (RCVDS) primitive.
- SE.UDF is the start of the undefined portion (message area).

The content of a packet as obtained from the free-packet pool is undefined.

The global symbol QE.LEN represents the length of the undefined portion, in bytes; the symbol SE.SIZ represents the overall packet size. These symbols, and the SE.xxx offset symbols shown above, are defined by the QUEDFS system macro.

### 2.2.3 System Queues

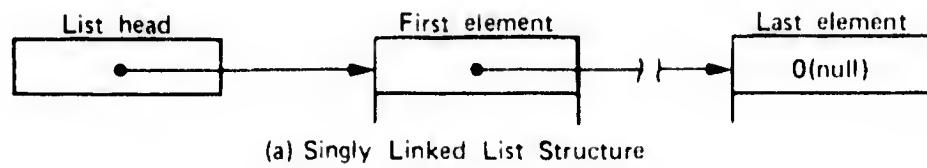
The kernel maintains a number of queues -- ordered lists -- of dynamically related elements such as PCBs or message packets. Two different queuing mechanisms are used: the singly linked list with either FIFO or priority ordering of elements and the doubly linked list.

## PROCESSES AND SYSTEM DATA STRUCTURES

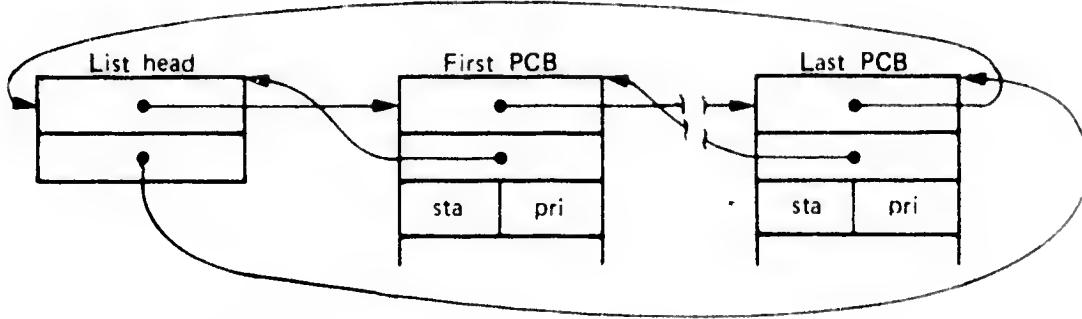
2.2.3.1 **Singly Linked Lists** - A singly linked list structure uses one link word (or pointer) per list element and is used for the following purposes:

- The blocking queue of a semaphore or a ring buffer (waiting process list)
- The packet queue of a queue semaphore
- The list of all current processes (all PCBs)
- The fork request queue (an internal queue associated with ISR management)
- The exception-handler dispatch lists (internal queues associated with exception dispatching)
- The static process list (used by INIT)
- The free-memory lists
- Structure name table lists
- Kernel resumption list

A singly linked list is shown schematically in Figure 2-11(a). The list head consists of a single link word. The link word of a list element may or may not be the first word of the element, but in all cases except the kernel resumption list, the link word points to the beginning of the successor element. The list is terminated by a zero value.



(a) Singly Linked List Structure



(b) Doubly Linked List Structure (State Queue)

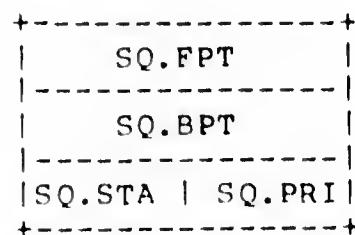
Figure 2-11 System Queue Structures

A singly linked list may be either FIFO or priority-ordered, depending on the ordering attribute associated with the list.

## PROCESSES AND SYSTEM DATA STRUCTURES

**2.2.3.2 Doubly Linked Lists** - A doubly linked list structure uses two link words -- a forward pointer and a backward pointer -- per list element. This list structure is used for the ready-active and ready-suspended state queues, where insertion or extraction of a PCB at any point in the queue is a frequently performed operation. The backward pointer makes such operations much faster, eliminating the queue traverse otherwise needed to determine the predecessor element. The ready-active state queue is priority-ordered; the ready-suspended state queue is, arbitrarily, LIFO-ordered.

A doubly linked list is shown schematically in Figure 2-11(b). The list head has the following format, which is identical to the first three words of a PCB:



In the format above:

- SQ.FPT is the forward link word, the pointer to the first PCB.
- SQ.BPT is the backward link word, the pointer to the last PCB.
- SQ.PRI is reserved.
- SQ.STA is the state code (either SC.RDA or SC.RDS).

The list is terminated by pointing its last element (PCB) back to the list head. If a state queue is empty, both link words of the list head point back to the list head.

### 2.2.4 System-Common Memory Organization

All dynamic system data structures are allocated in the system-common memory area. This area, beginning at \$FREE, normally constitutes the major portion of the kernel's impure-data segment; see Figure 2-5. The size of this area is determined at system-build time by the RESOURCES configuration macro. The rest of the kernel data segment consists of the following:

- System-interrupt stack (beginning at \$KSTKL)
- Kernel's private impure data (beginning at \$REDYS)
- Interrupt dispatch block area (beginning at \$IDBEG)

(These areas precede the system-common area.)

## PROCESSES AND SYSTEM DATA STRUCTURES

System-common memory is subdivided by the system-initialization (INIT) routine as follows:

- The free-memory pool, from which all system data structures other than queue packets are allocated. The size of this pool is determined by the structures parameter of the RESOURCES macro. The default size is 1000 bytes. After establishing this pool, the INIT routine creates the static-process PCBs.
- The free-packet pool, from which processes obtain "empty" packets via the ALPK\$ or ALPCS primitives. The INIT routine preallocates n packets in this pool, where n is the number of packets requested in the packets parameter of the RESOURCES macro. The default is 20 packets. Thus, the size of this pool is  $n * SE.SIZ$  bytes; the standard value of SE.SIZ is 40 (see Section 2.2.2). The packets are linked into a free-element queue by INIT.

The structure of the free-memory pool can be described as follows. Blocks of memory are allocated from the free-memory pool as data structures are created and are deallocated -- returned to the pool -- when structures are deleted, by the common kernel procedures \$ALLOC and \$DALOC. (These procedures are used only by primitives operations and other kernel routines.)

The allocation/deallocation algorithms assume that free memory is linked together in a singly linked, open list structure, with the first word of a memory block used as a pointer to the next available block and the second word used to indicate the size of the block in bytes. Thus, the free-memory pool looks as shown in Figure 2-12.

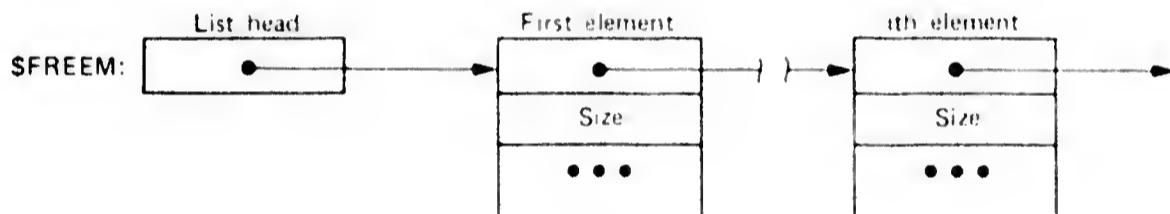


Figure 2-12 Free-Memory Pool

A zero pointer value terminates the list. The INIT routine initializes the kernel variable \$FREEEM to point to the first word of the free-memory pool. The initial size of the pool is placed in the second word, the first word being an empty (zero) pointer to the next entry in the list.

Memory is allocated from the pool in multiples of four bytes. The allocation algorithm is first-fit. If the first element that can accommodate a given request is larger than the amount of space requested, the space is allocated from the end of the element. Since structures are usually created and deleted in an arbitrary sequence, the free-memory pool can become fragmented during system operation.

The \$DALOC procedure returns a released memory block to the free-memory pool. Whenever possible, \$DALOC will merge contiguous memory elements into a single element during deallocation.

## CHAPTER 3

### MACRO-11 PRIMITIVE SERVICE REQUESTS

This chapter describes the MACRO-11 interface to the real-time primitive services provided by the MicroPower/Pascal kernel. This chapter describes the purposes and applications of the kernel primitives, as well as the detailed syntax and semantics of the macro calls used to request the primitive services. In addition, the chapter provides information about structure descriptor blocks and process descriptor blocks, which are commonly used in conjunction with primitive service requests. For ease of reference, the individual primitive descriptions are in alphabetic order, by primitive name.

The MACRO-11 interface consists of a set of keyword macros. These macros facilitate construction of the argument block required by each kernel primitive routine, as well as invocation of the routine. Three forms of macro call are provided for each primitive service. The three variants permit, respectively:

1. Runtime construction or modification of the argument block in user-specified RAM storage
2. Runtime construction of the argument block on the user's stack
3. Assembly-time construction of the argument block in either ROM or RAM storage

The MicroPower/Pascal compiler also provides an interface to the primitive services described in this chapter. This interface consists of the predefined procedure and function calls known collectively as the MicroPower/Pascal real-time programming extensions. Each macro call description in this chapter includes the name of the equivalent Pascal procedure or function.

#### 3.1 GENERAL CONVENTIONS AND USAGE RULES

Kernel primitives are invoked from process level by the IOT trap instruction, which in MicroPower/Pascal is reserved solely for this purpose. The IOT instruction is followed immediately by the global entry point symbol for the desired primitive, of the form \$prim, as follows:

```
IOT  
$prim
```

with R0 pointing to the caller's argument block. The primitive service macro calls generate this sequence as part of their expansion.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

The primitive name (prim above) is always a 4-character mnemonic for the service performed by the primitive; for example, CRST for the Create Structure primitive and SGNL for the Signal Semaphore primitive. The corresponding macro call names are formed by appending \$, SS, or SP to the mnemonic; for example, CRSTS, CRSTSS, or CRSTSP. The suffixes \$, SS, and SP identify variant forms of the basic prim macro call; the three variants provide maximum coding flexibility and efficiency. The three variants differ as follows:

1. The variant prim\$ is used for dynamic generation of modification of both the required argument block in a caller-specified memory area and the IOT sequence. (This variant can also be used in a special form to pass a preexisting argument block that may have been built in ROM with the prim\$P macro variant.)
2. The variant prim\$S is used for dynamic generation of both the required argument block on the user's stack and the IOT sequence.
3. The variant prim\$P is used for assembly-time generation, in either ROM or RAM, of the argument block only. (No IOT sequence is generated.) This variant is used in conjunction with the null- or single-argument form of the prim\$ variant, as described below.

The following subsections describe the general form of each macro variant and the usage rules associated with each.

### 3.1.1 Macro Variant prim\$

**General Form** - A primitive that takes N arguments will have a corresponding prim\$ macro call of the general form:

```
prim$ area,argument_1,argument_2,...,argument_N
```

This macro call expands into a code sequence of the general form:

```
MOV area,R0  
MOV argument_1,(R0)  
MOV argument_2,2(R0)  
:  
:  
MOV argument_N,N*2(R0)  
IOT  
$prim
```

Various optimizations of this sequence are produced for special cases. For example, a call with a relatively large number of arguments produces the following:

```
MOV area,R0  
MOV R0,-(SP)  
MOV argument_1,(R0)+  
MOV argument_2,(R0)+  
:  
:  
MOV argument_N,(R0)+  
MOV (SP)+,R0  
IOT  
$prim
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

If one or more of the primitive argument values are null in the call, the corresponding move instructions are omitted in the expansion. For example, a call may have the form:

```
prim$ area,,argument_2,,argument_4
```

This call produces the following expansion:

```
MOV area,R0  
MOV argument_2,2(R0)  
MOV argument_4,6(R0)  
IOT  
$prim
```

Similarly, a call may have the form:

```
prim$ area
```

This call produces the following expansion:

```
MOV area,R0  
IOT  
$prim
```

Note that this expansion allows modification of selected fields in an existing argument block or use of an existing argument block without modification.

### NOTE

If the area parameter is null, it is assumed that R0 already contains the address of the argument block, and the MOV area,R0 instruction is omitted in the expansion. Therefore, if the entire argument list is missing, the macro expansion produces only the IOT sequence.

**Usage Rules** - As implied by the foregoing, the general usage rules for the prim\$ form of macro call are the following:

1. If any of the second through Nth macro arguments are null, the precall content of the corresponding argument block location is not modified by the call.
2. If the area parameter is null, the argument block address must be stored in R0 prior to the call.

### 3.1.2 Macro Variant prim\$\$

**General Form** - A primitive that takes N arguments will have a corresponding prim\$\$ (stack version) macro call of the general form:

```
prim$$ argument_1,argument_2,...,argument_N
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

This macro call expands into a code sequence of the general form:

```
MOV argument_N,-(SP)
MOV argument_N-1,-(SP)
.
.
.
MOV argument_1,-(SP)
MOV SP,R0
IOT
$prim
< pop arguments from stack >
```

The argument list may be omitted, as in call of the form:

```
prim$S
```

This call produces the following degenerate expansion:

```
MOV SP,R0
IOT
$prim
```

Note that this expansion assumes that an appropriate argument block exists on the stack when the call is executed.

**Usage Rules** - The general usage rules for the prim\$S form of macro call are the following:

1. If one macro argument is specified, all arguments must be specified. The stack is purged of all arguments on return from the call.
2. If no macro argument is specified, the desired argument block must be constructed on the stack prior to the call. The stack is not purged following the call.

### 3.1.3 Macro Variant prim\$P

**General Form** - A primitive that takes N arguments will have a corresponding prim\$P (parameters only) macro call of the general form:

```
[label:] prim$P argument_1,argument_2,...,argument_N
```

This macro call expands into a code sequence of the general form:

```
[label:] .WORD argument_1
        .WORD argument_2
        .
        .
        .
        .WORD argument_N
```

If one or more of the macro arguments are null in the call, a 0 is generated for that argument. For example, a call may have the form:

```
[label:] prim$P ,argument_2,,argument_4
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

This call produces the following expansion:

```
[label:] .WORD 0
        .WORD argument_2
        .WORD 0
        .WORD argument_4
```

**Usage Rule** - If an argument is null in the macro call, the corresponding location in the argument block will have a zero value.

**Guidelines** - The prim\$P macro variant can be used within the scope of a PDAT\$ macro to generate an argument block in ROM storage. (This assumes that the argument values will never be modified and that the primitive operation to which the block is passed does not return any values in the block.) The "prim\$ area" form of macro call can be used to pass the address of the block to the appropriate primitive.

Alternatively, the prim\$P macro can be used within the scope of an IMPUR\$ macro to allocate an argument block area of the required size in RAM storage. (The argument list is not needed for this purpose.) The argument block is then filled in at runtime prior to issuing the IOT.

### 3.1.4 Error Returns

An error condition encountered by a primitive service routine is reported to the caller by an immediate return of control to the call site, with the carry (C) bit set in the program status word. An error code is returned in R0.

Global symbols (of the form E.xxxx) map the numeric error code values returned in R0. These symbols are defined by the ERRDF\$ macro. The error codes indicate the type of error that occurred: E.IIID for an invalid structure or process identifier, E.ILPM for an illegal parameter value, and E.BUSY for a busy or otherwise unavailable resource, for example. The particular error codes returned by a given primitive are specified in the description of that primitive.

#### NOTE

Some primitives also return a nonerror function value in R0. In this case, the C bit is cleared on return from the primitive.

### 3.1.5 Structure Descriptor Block (SDB) Usage

A structure descriptor block (SDB) describes a particular semaphore or ring buffer. (These system structures are allocated in kernel space, but are created, used, and deleted at user request.) Approximately 20 primitives act directly on a given semaphore or ring buffer; therefore, the structure must be identified (indirectly) in the corresponding primitive request.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

An example of such a request is the Signal Semaphore (SGNL\$) primitive call, which has the form:

```
SGNL$ area,sdb
```

The sdb argument, a pointer to the SDB, indirectly identifies the semaphore to be signaled.

The user allocates and initializes an SDB in process space. An SDB for a named structure is a 6-word block consisting of a 3-word structure identifier -- filled in by the kernel -- and a 6-byte alphanumeric structure name. (The SDB for an unnamed structure can be abbreviated to four words, as explained further on.)

The format of an SDB is as follows:

Structure index	Structure ID (kernel-level identifier)
Structure	
serial number	
6-character	
structure	Global process-level identifier
name	

An SDB must be in RAM and may be constructed on the stack.

An SDB has two uses, as follows:

1. To specify the name, if any, of a structure to be created by the CRSTS primitive
2. To access, through other primitive services, an existing structure that is referenced by either structure ID or structure name

When a structure is either created (by CRSTS) or accessed by structure name (by any other primitive), the primitive writes a structure identifier into the first three words of the SDB. In subsequent uses of the filled-in SDB, the structure identifier permits direct, optimized access to the structure, bypassing the table-lookup step needed for a reference by name. This use results in faster processing of the primitive request.

Primitives that operate on structures test the first word of the passed SDB (the structure index) to determine how to use information contained in the SDB. If the index value is nonzero, the primitive assumes that the structure-ID field contains valid information and uses it to locate the structure. In this case, the last three words of the SDB are not significant. If the index value is 0, a reference by name is implied, and the primitive uses the contents of the structure name field to find the structure by a name-table lookup procedure. (The latter case is invalid for an unnamed structure.)

Structure and process names must be unique throughout an entire application system. (A process name describes a process control block, another kind of system structure.)

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**3.1.5.1 Initialization of SDBs for Named Structures** - Before using an SDB to either create a named structure or refer to an existing structure by name (in a SGNLS or WAITS request), you must initialize the SDB as follows:

1. The value of the first word (structure index) must be 0.
2. The structure name field must contain the ASCII character string used to globally name the structure. If shorter than six characters, the name string must be left-justified in the field; the trailing character positions should be space-filled.

By system convention, a structure name shorter than six printing characters is padded with trailing spaces. Any unused high-order bytes of the structure name field in the SDB must contain the ASCII SPACE character (octal 040). (The MicroPower/Pascal compiler space-fills such names by default.) This convention is significant, for example, if you construct an SDB to describe a semaphore created by a system service process, such as the I/O request queue semaphores established by the standard device handlers. (The handler request queues have names of the form \$xxx followed by two spaces.)

**3.1.5.2 Initialization of SDBs for Unnamed Structures** - An unnamed structure is created by passing the CRSTS primitive a pointer to an SDB that contains a 0 in the first byte of the structure name field. Since the last five bytes of the SDB are not significant in this case, an SDB for an unnamed structure need be only seven bytes long.

Before using an SDB to create an unnamed structure, you must initialize the SDB as follows:

1. The value of the first word (structure index) must be 0.
2. The value of the seventh byte (first byte of the structure name field) must be 0.

For subsequent references to the structure, only the first three words of the SDB -- the structure ID field -- are needed, as is the case with named structures.

To refer to an existing unnamed structure, the calling process must supply an SDB containing a valid structure identifier. Therefore, in order to access such a structure, a process other than the creator must also have access to the SDB used to create it. In a mapped environment, an unnamed structure can be shared only among processes that reside in the same address space. (The SDB could be sent as a message to another process, however.)

### **3.1.6 Process Descriptor Block (PDB) Usage**

A process descriptor block (PDB) describes a particular process. (The PDB identifies a process control block, the system structure in which the state and context of an existing process are stored.) Like an SDB,

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

a PDB is a 6-word block consisting of a 3-word process identifier filled in by the kernel and a 6-byte alphanumeric process name. The format of a PDB is as follows:

Process index	Process ID
Process	(kernel-level identifier)
serial number	
6-character	
process	Global process-level identifier
name	

The use of a PDB for process creation (CRPCS primitive) is identical to the use of an SDB for structure creation, as described above.

The primitives that operate on existing processes, however, provide a shorthand way for the calling process to specify itself as the process to be acted on, as well as some other process. Thus, the rules for reference to an existing process differ from those for reference to an existing structure. The use of a PDB in requests that operate on processes is identical to the use of SDBs in other requests. However, if the PDB argument is 0, the request will operate on the current process.

Reference to an existing process is best described by example. The primitives that operate on specified processes are the following:

- GTSTS (Get Process Status)
- SPNDS (Suspend Process)
- RSUM\$ (Resume Process)
- STPCS (Stop Process)

Each of these primitives requires the address of a PDB as a calling argument and interprets this argument in the same manner. For example, the call for the Get Process Status primitive, which returns information about a given process, is of the form:

```
GTSTS area,pdb buf
```

The address of the PDB that identifies the subject process is `pdb`, and `buf` points to the caller's information-return buffer.

The primitive interprets the `pdb` argument value as follows:

1. If the argument value is 0, indicating no PDB, the primitive assumes that the calling process is to be acted on. (In the case of GTSTS, status information about the calling process is returned to the caller.) If the argument value is nonzero, the primitive uses the indicated PDB to locate the process to be acted on, as follows.

#### **MACRO-11 PRIMITIVE SERVICE REQUESTS**

2. If the process index value of the PDB is nonzero, the primitive uses the contents of the process ID field to locate the process to be acted on. If the process index value is 0, the primitive examines the process name field of the PDB.
3. If the value of the first byte of the name field is 0, the primitive assumes that the calling process is to be acted on. If the value is nonzero, the primitive uses the process name string to locate the process to be acted on.

In all cases, if a PDB address is specified in the call and if the process index value is 0, the primitive writes a valid process identifier in the process ID field during the primitive operation. This is analogous to the action performed for a structure access by structure name, as described in Section 3.1.5, and permits optional access to the process on subsequent uses of the PDB.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.2 ALPCS (CONDITIONALLY ALLOCATE PACKET)

Pascal equivalent: COND\_ALLOCATE\_PACKET Function

The Conditionally Allocate Packet (ALPCS) primitive allocates a message packet (standard queue element) from the kernel's free-packet pool if one is available or returns a FALSE indication if not. If a free packet is available, it is logically removed from the free-packet pool. A pointer to the packet is returned to the caller, and the kernel-defined value TRUE is returned in R0. If all packets are in use at the time of the call, the primitive returns control immediately with the kernel-defined value FALSE in R0.

This primitive permits the caller to obtain a packet pointer for use in conjunction with either the Signal Queue Semaphore (SGLQS) or the Conditionally Signal Queue Semaphore (SGQCS) primitive operation, without blocking if the request cannot be satisfied. (Compare with ALPKS, the unconditional form.)

The DAPKS primitive is the inverse of ALPCS, allowing a process to deallocate a message packet.

#### Syntax

The three variants of the ALPCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
ALPCS	ALPCS [area,qelm]
ALPCSS	ALPCSS [qelm]
ALPCSP	ALPCSP

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

qelm	The address of a location that is to receive the packet pointer returned by the primitive. This argument has the form:
------	--

[QELM=]destination-address

Or, it may be null. (If specified, it must be a word address.)

If the qelm argument is null, the packet pointer returned by the primitive is available only in the calling argument block. If the argument is null in the stack (SS) version of the macro call, the returned pointer value is left on the stack. (Ordinarily, the argument block is purged from the stack following the call.) In the parameters-only (SP) version of the macro call, no qelm argument is specified, and the returned pointer value is available only in the calling argument block. (See the Restrictions section below.)

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Restrictions** - The argument block must be in read/write memory.

You can use the parameters-only (SP) version of the macro call in a RAM-only system, provided that you correctly access the queue element word in the argument block. However, you cannot use the SP call in either the ROM or RAM portion of a ROM/RAM system.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the ALPCSx macro has the following format:

```
R0 -> +-----+  
          | --- | <- Default destination of  
          +-----+     returned pointer value
```

### Semantics

The ALPCS primitive tests the free-packet pool for a free packet. If the pool contains at least one packet, the primitive logically removes a packet from the pool and returns the address of that packet in the argument block, from which it is moved to a user-specified location by the macro expansion if requested (qelm argument). The primitive also returns the kernel-defined value TRUE in R0.

If there are no free packets, the primitive returns immediately to the calling process with the kernel-defined value FALSE in R0.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

### Error Returns

None.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.3 ALPKS (ALLOCATE PACKET)

Pascal equivalent: ALLOCATE\_PACKET Procedure

The Allocate Packet (ALPKS) primitive allocates a message packet (standard queue element) from the kernel's free-packet pool. If a free packet is available, it is logically removed from the free-packet pool, and a pointer to the packet is returned to the caller. If all packets are in use at the time of the call, the calling process is blocked until the request can be satisfied. (If several processes are concurrently waiting for packet allocation, the requests are satisfied according to process priority as packets are returned to the pool.)

This primitive permits the caller to obtain a packet pointer for use in conjunction with either the Signal Queue Semaphore (SGQ\$) or the Conditionally Signal Queue Semaphore (SGQS\$) primitive operation.

The Conditionally Allocate Packet primitive (ALPC\$) permits a process to request packet allocation without blocking if no packets are free.

The inverse of ALPKS is the DAPKS primitive, allowing a process to deallocate a message packet.

#### Syntax

The three variants of the ALPKS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
ALPKS	ALPKS [area,qelm]
ALPKSS	ALPKSS [qelm]
ALPKSP	ALPKSP
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
qelm	The address of a location that is to receive the packet pointer returned by the primitive. This argument has the form:  [QELM=]destination-address  Or, it may be null. If specified, it must be a word address. Note that the argument you specify is expanded directly into a MOV instruction destination argument; therefore, it should not contain an immediate expression indicator (#).

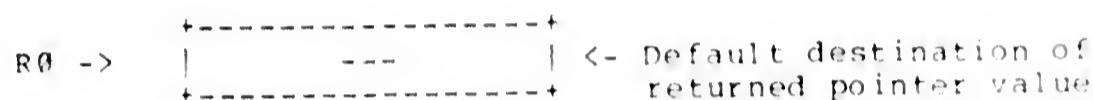
## MACRO-11 PRIMITIVE SERVICE REQUESTS

If the qelm argument is null, the packet pointer returned by the primitive is available only in the calling argument block. If the argument is null in the stack (SS) version of the macro call, the returned pointer value is left on the stack. (Ordinarily, the argument block is purged from the stack following the call.) In the parameters-only (SP) version of the macro call, no qelm argument is specified, and the returned pointer value is available only in the calling argument block. (See the Restrictions section below.)

**Restrictions** - The argument block must be in read/write memory.

You can use the parameters-only (SP) version of the macro call in a RAM-only system, provided that you correctly access the queue element word in the argument block. However, you cannot use the SP call in either the ROM or RAM portion of a ROM/RAM system.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the ALPKSx macro has the following format:



### Semantics

The ALPKS primitive tests the free-packet pool for a free packet. If the pool contains at least one packet, the primitive logically removes a packet from the pool and returns the address of that packet in the argument block. If requested (qelm argument), the macro expansion moves the address to a user-specified location.

If there are no free packets, the primitive blocks the calling process on a semaphore associated with the free-packet pool and calls the scheduler. The process remains on the semaphore's waiting process list, in priority order relative to other processes that may also be waiting, until enough packets have been freed to permit allocation. (See the DAPKS primitive.)

### Error Returns

None.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.4 CCNDS (CONNECT TO EXCEPTION CONDITION)

Pascal equivalent: CONNECT\_EXCEPTION Procedure  
DISCONNECT\_EXCEPTION Procedure

The Connect to Exception Condition (CCNDS) primitive establishes a process as the exception handler for a particular group of processes and for a specified type of exception. (The handler is, presumably, a privileged process.) The primitive establishes an existing queue semaphore, supplied by the caller, as the exception queue through which the specified exceptions will be signaled by the kernel.

This primitive allows a process to be activated when a specific type of exception occurs in any of the processes belonging to the specified exception group. The handler receives the exception by doing a WAIQS operation on its exception queue semaphore.

The handler can call the CCNDS primitive several times to specify either the same exception type for several process groups or several exception types for one process group. Alternatively, a process can establish itself as an exception handler for all process groups -- all processes in the system, regardless of exception-group code.

The PCB of the process causing the exception is placed on the handler's exception queue, in exception wait state, when the queue semaphore is signaled by the kernel. The handler must then process the exception condition and dispose of the PCB by using the Dismiss Exception Condition (DEXCS) primitive. The DEXCS primitive allows three basic courses of action: abort the process, pass exception to the process, or return the process to ready state (that is, cancel the exception).

#### Syntax

The three variants of the CCNDS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CCNDS	CCNDS [area,mask,group,sdb]
CCNDSS	CCNDSS [mask,group,sdb]
CCNDSP	CCNDSP [mask,group,sdb]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
mask	The type of exception, as indicated by predefined bit-mask symbols, to be serviced by the handler process. The exception-type symbols are defined by the EXMSKS macro and are described in Chapter 7. This argument has the form:  [MASK=]symbol

## MACRO-11 PRIMITIVE SERVICE REQUESTS

group

The group of processes, as indicated by an integer group code value of 0 to 255, for which exception conditions will be serviced. (See the group argument of the CRPCS\$ and DFSPCS macros.) This argument has the form:

[GROUP=]integer-value

The group code 0 is the wildcard group code, indicating all exception-handling groups.

sdb

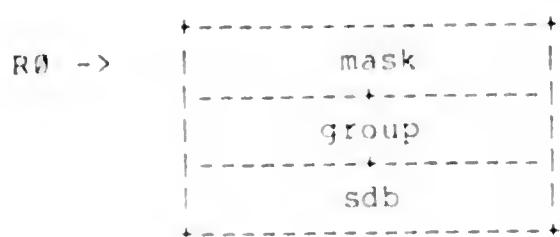
The address of a structure descriptor block (SDB) that identifies the queue semaphore to be used as the exception queue. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=]sdb-address

### NOTE

If the sdb argument value is 0, the meaning of the request changes to "disconnect exception handler" for the specified exception type and process group. That is, the exception queue that was connected by a previous call specifying the same exception condition and group is disconnected from that particular type/group combination.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the CCNDSx macro has the following format:



### Semantics

The CCNDS primitive makes an entry -- in the kernel's exception-dispatching table -- that describes a queue semaphore, identified by the passed SDB, for a given combination of exception type and process group or groups. The primitive then returns to the caller.

When an exception condition of the specified type occurs in a process belonging to the specified group, the semaphore established as the exception queue for that particular combination is signaled by the kernel's exception dispatcher. The handler process waiting on that semaphore is unblocked. (Chapter 7 describes exception dispatching in detail.)

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

The queue element passed by the kernel via the exception queue semaphore is the PCB of the process that caused the exception (rather than a standard packet). The faulting process is in the exception-wait state. (Note that the handler must be a privileged process in order to access the information contained in the PCB. After processing the exception, the handler must return the PCB to one of the kernel's state queues by using the DEXCS call.)

The group code permits several exception handlers for the same exception condition to coexist, each handler implementing a management strategy suited to a given group of processes. If no handler exists for a group of processes and if a member of that group causes an exception, the kernel either passes the exception to the exception entry point of the faulting process or aborts the process by forcing its termination entry point. (See the SERAS primitive concerning exception handling within the process.)

### **Error Returns**

**E.IILPM** - illegal parameter specified; the mask word was either all 0 bits, or more than one bit was set.

**E.BUSY** - Either the kernel's free-memory pool was exhausted -- that is, could not allocate space for the connection -- or a semaphore was already connected to the specified group/condition.

### **Implementation Note**

If one exception-management strategy is applicable to several groups for a particular exception type, several CCNDS calls can be used to connect one type of exception from several process groups to the same exception queue. Alternatively, several CCNDS calls can be used to connect several different types of exceptions from one process group to the same exception queue. If a handler manages more than one exception condition, it can create individual subprocesses. The handler in turn signals these subprocesses according to exception type. Thus, each subprocess asynchronously manages a single exception condition.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.5 CHGP\$ (CHANGE PROCESS PRIORITY)

Pascal equivalent: CHANGE\_PRIORITY Procedure

The Change Process Priority (CHGP\$) primitive changes the priority of the calling process to the value specified in the call. Thus, the calling process can dynamically modify its scheduling priority, normally to a lower value.

This primitive allows a process to lower its priority to a normal operating level -- less than 248 -- after starting at a high priority level, if necessary, for initialization. The special start-up priorities are 248 to 255. The highest start-up priority, 255, is used by a static process that must execute a one-time initialization sequence involving creation of global system structures and, possibly, subprocesses as well. Other processes may use start-up priorities in the range 248 to 254 to ensure a particular starting order among a group of related processes.

The initialization code of a given static process might, for example, create a queue semaphore that must exist before another process begins execution at its normal operating priority, which may, in fact, be higher than that of the process that must create the semaphore. The initialization code would end with a CHGP\$ request to lower priority to an appropriate level. In general, global system structures must be created at a priority level that is higher than any normal operating priority used in the system, in order to prevent start-up race conditions among processes in different process families.

#### Syntax

The three variants of the CHGP\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CHGP\$	CHGP\$ [area,pri]
CHGP\$S	CHGP\$S [pri]
CHGP\$P	CHGP\$P [pri]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
pri	The new scheduling priority value for the calling process. This argument has the form:  [PRI=]priority-value  The value must be within the range 0 to 255.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the CHGP\$X macro has the following format:



### Syntax Example

```
CHGP$S #125 ; Note: CHGP$S 125 would be invalid!
```

### Semantics

The CHGP\$ primitive places the specified priority value in the PC.PRI field of the caller's PCB and calls the scheduler. This will cause the calling process to be preempted if a higher-priority process is ready active at the time of the call. Otherwise, control returns to the calling process.

### Error Returns

E.ILPM - Invalid parameter; the specified priority value is less than 0 or greater than 255.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.6 CINT\$ (CONNECT TO INTERRUPT)

Pascal equivalent: CONNECT\_INTERRUPT Procedure  
Pascal variant: CONNECT\_SEMAPHORE Procedure

The Connect to Interrupt (CINT\$) primitive associates an interrupt vector with an interrupt service routine (ISR) entry point specified in the call.

The CINT\$ primitive allows a process to establish itself as a device handler and allows the process to define the ISR code segment. The CINT\$ primitive is normally used only by a process with DRIVER mapping type, in the case of a mapped environment.

#### Syntax

The three variants of the CINT\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

#### Variant

#### Macro Call

CINT\$

CINT\$ [area,vec,ps,val,imp,isr,pic]

CINT\$S

CINT\$S [vec,ps,val,imp,isr,pic]

CINT\$P

CINT\$P [vec,ps,val,imp,isr,pic]

#### Parameter

#### Definition

area

The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

vec

The address of the hardware interrupt vector to be connected to the ISR. This argument has the form:

[VEC=]vector-address

ps

The content of the PSW desired on dispatch to the ISR. This argument sets the processor priority level at which the ISR is to execute when entered. If priority level 7 is requested -- that is, PS = 340(octal) -- a special form of ISR dispatching is implied, as described in Chapter 8. (The CC bits can also be set with this argument, but not the T bit.) This argument has the form:

[PS=]word-value

The effective PSW value is in the low byte.

val

An arbitrary value to be passed to the ISR in R4 on interrupt dispatch. (Typical uses of val are to pass a device address, table index, or other means of identifying the vector causing the

## MACRO-11 PRIMITIVE SERVICE REQUESTS

interrupt, in the case of an ISR connected to several vectors.) This argument has the form:

[VAL=]word-value

**imp** In an unmapped system, an arbitrary value to be passed to the ISR. In mapped systems, if the PIC argument (see below) is a 1, this value is assumed to be an address, which is converted to a PAR 3 value and is passed to the ISR in R3 on interrupt. If the PIC argument is a 0, the value is passed to the ISR unchanged. This parameter is typically used to pass the address of the ISR's impure area. This argument has the form:

[IMP=]impure-area-address

**ist** The address of the ISR code segment. This argument has the form:

[ISR=]isr-address

**pic** A Boolean value indicating that the ISR is implemented as non-PIC code (0) or PIC code (1). (PIC stands for Position Independent Code.) This argument has the form:

[PIC=]word-value

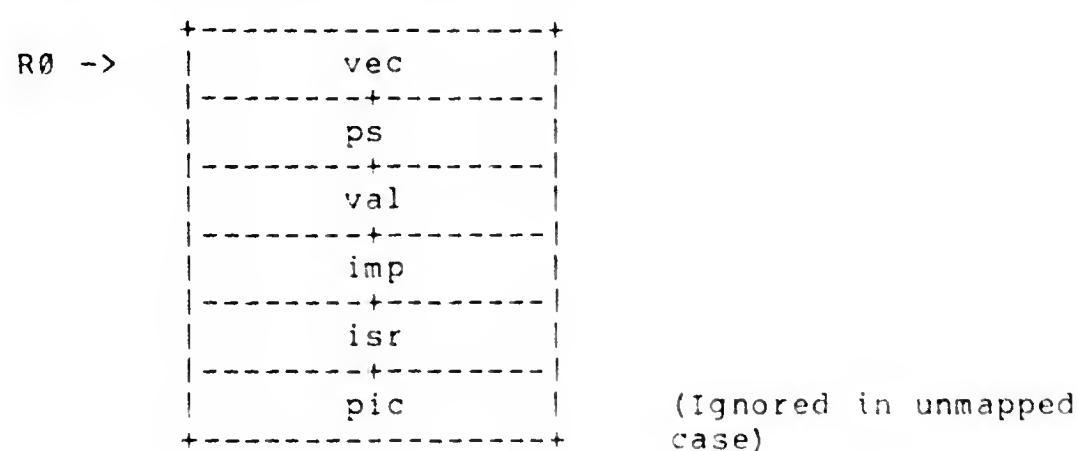
The value may be 0 or 1. This argument is ignored in an unmapped system. (PIC code is typically used only by a process that does not have DRIVER mapping.)

**Restrictions** - The ISR code segment must not exceed 8129 bytes in length.

The ISR's impure-data segment must not exceed 8129 bytes in length.

In a mapped environment, if PIC coding is not used, the ISR code- and data-segment virtual addresses must be relocated -- at build time -- to fall within the PAR 2 and PAR 3 address ranges.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the CINT\$X macros has the following format:



## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.6 CINT\$ (CONNECT TO INTERRUPT)

Pascal equivalent: CONNECT\_INTERRUPT Procedure  
Pascal variant: CONNECT\_SEMAPHORE Procedure

The Connect to Interrupt (CINT\$) primitive associates an interrupt vector with an interrupt service routine (ISR) entry point specified in the call.

The CINT\$ primitive allows a process to establish itself as a device handler and allows the process to define the ISR code segment. The CINT\$ primitive is normally used only by a process with DRIVER mapping type, in the case of a mapped environment.

#### Syntax

The three variants of the CINT\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CINT\$	CINT\$ [area,vec,ps,val,imp,isr,pic]
CINT\$S	CINT\$S [vec,ps,val,imp,isr,pic]
CINT\$P	CINT\$P [vec,ps,val,imp,isr,pic]

#### Parameter      Definition

area      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

vec      The address of the hardware interrupt vector to be connected to the ISR. This argument has the form:

[VEC=]vector-address

ps      The content of the PSW desired on dispatch to the ISR. This argument sets the processor priority level at which the ISR is to execute when entered. If priority level 7 is requested -- that is, PS = 340(octal) -- a special form of ISR dispatching is implied, as described in Chapter 8. (The CC bits can also be set with this argument, but not the T bit.) This argument has the form:

[PS=]word-value

The effective PSW value is in the low byte.

val      An arbitrary value to be passed to the ISR in R4 on interrupt dispatch. (Typical uses of val are to pass a device address, table index, or other means of identifying the vector causing the

## MACRO-11 PRIMITIVE SERVICE REQUESTS

interrupt, in the case of an ISR connected to several vectors.) This argument has the form:

[VAL=]word-value

**imp** In an unmapped system, an arbitrary value to be passed to the ISR. In mapped systems, if the PIC argument (see below) is a 1, this value is assumed to be an address, which is converted to a PAR 3 value and is passed to the ISR in R3 on interrupt. If the PIC argument is a 0, the value is passed to the ISR unchanged. This parameter is typically used to pass the address of the ISR's impure area. This argument has the form:

[IMP=]impure-area-address

**isr** The address of the ISR code segment. This argument has the form:

[ISR=]isr-address

**pic** A Boolean value indicating that the ISR is implemented as non-PIC code (0) or PIC code (1). (PIC stands for Position Independent Code.) This argument has the form:

[PIC=]word-value

The value may be 0 or 1. This argument is ignored in an unmapped system. (PIC code is typically used only by a process that does not have DRIVER mapping.)

**Restrictions** - The ISR code segment must not exceed 8129 bytes in length.

The ISR's impure-data segment must not exceed 8129 bytes in length.

In a mapped environment, if PIC coding is not used, the ISR code- and data-segment virtual addresses must be relocated -- at build time -- to fall within the PAR 2 and PAR 3 address ranges.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the CINTSx macros has the following format:

R0 ->	+-----+   vec   +-----+   ps   +-----+   val   +-----+   imp   +-----+   isr   +-----+   pic   +-----+   (Ignored in unmapped   case)
-------	---

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
CINT$ AREA=#CAREA,VEC=1300,PS=1200,VAL=10,IMP=#DATA,ISR=#DEV1_R,  
PIC=#0
```

### Semantics

The CINT\$ primitive sets up the interrupt dispatch block (IDB) associated with the specified vector, causing interrupts to be dispatched to the specified ISR entry point. The primitive also identifies the caller as the process owning the connected vector (compare with the DINT\$ primitive).

Chapter 8 contains information closely related to the use of CINT\$ and the coding of ISRs. It describes interrupt dispatching, which is affected by certain CINT\$ arguments -- especially the PS and IMP values -- and describes the kernel/ISR interface in general.

### Error Returns

- E.ILVC - Illegal vector specified (outside valid vector range V.MIN - V.MAX).
- E.BUSY - Vector already in use.
- E.ADDR - Invalid address (mapped systems only); invalid ISR mapping.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.7 CRPCS (CREATE PROCESS)

Pascal equivalent: Process-invocation statement

The Create Process (CRPCS) primitive service creates a dynamic process, as requested by the caller, and places it in the ready-active state, eligible for scheduling. This primitive permits an existing process, static or dynamic, to create and activate a subprocess. The created process will have a combination of the process attributes specified in the service request -- for example, priority, exception group -- and the attributes inherited from the parent process -- mapping, mapping type, exception routine address, and exception bit mask.

The CRPCS primitive constructs a process control block (PCB) for the new process. This PCB physically represents the process within the kernel, as described in Chapter 2.

Note that the Create Process service is transparent to the Pascal user; there is no predefined MicroPower/Pascal procedure equivalent to the CRPCS request. In Pascal, creation of a process is implicit in each call of a construct declared as a process.

#### Syntax

The three variants of the CRPCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CRPCS	CRPCS [area,pdb,pri,cxo.grp,ter, cxl,sti,stl,sth,start,ini]
CRPCSS	CRPCSS [pdb,pri,cxo.grp,ter,cxl, sti,stl,sth,start,ini]
CRPCSP	CRPCSP [pdb,pri,cxo.grp,ter,cxl, sti,stl,sth,start,ini]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
pdb	The address of the user-constructed process descriptor block (PDB) containing the name, if any, of the process to be created and in which the kernel returns information identifying the process. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:  [PDB=] pdb-address
pri	The priority value (0 to 255) to be associated with the process. This argument has the form:  [PRI=] integer-value

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**cxo** Any optional hardware context, as indicated by predefined bit-mask symbols, to be included -- saved and restored -- in the context switching performed for this process. The option symbols are:

CX\$FPP Floating-point registers

CX\$KT MMU registers

CX\$MCX Single memory location specified by cxl and intended for use primarily by the Pascal compiler

CX\$STD Standard context switching only -- that is, "save no additional context"

The option symbols may be ORed as required. These symbols are defined by the CXODFS macro. This argument has the form:

[CXO=]option[!option]

**grp** An arbitrary integer code value of 1 to 255 indicating the exception-handling group to which the process belongs. (See the Restrictions and Semantics sections below.) The exception group code values must be established by a convention agreed on by all process designers. This argument has the form:

[GRP=]integer-value

**ter** The entry point of the termination instruction sequence for the process. (See the Semantics section below.) This argument has the form:

[TER=]instruction-address-value

**cxl** The address of the user-memory location whose content is to be saved/restored when context switching this process. This argument is meaningful only if CX\$MCX is specified in cxo; otherwise, the argument value must be 0. This argument has the form:

[CXL=]address-value

**sti** The initial address of the user-allocated process stack. This argument has the form:

[STI=]top-of-stack-address

**stl** The address of the low boundary of the process stack. This argument has the form:

[STL=]low-bound-address

**sth** The address of the high boundary of the process stack. This argument has the form:

[STH=]high-bound-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**start**      The initial entry point for the process. This argument has the form:

[START=]first-instruction-address

**ini**      The initial value for the location cxl -- that is, the value to be stored in that location when the process is first executed. This argument is meaningful only if CX\$MCX is specified in cxo; otherwise, it may be null (in the CRPC\$ form only). This argument has the form:

[INI=]word-value

**Restrictions** - Exception group code 0 may not be specified for a process, and codes 128 to 255 (200 to 377 octal) are reserved for possible future use.

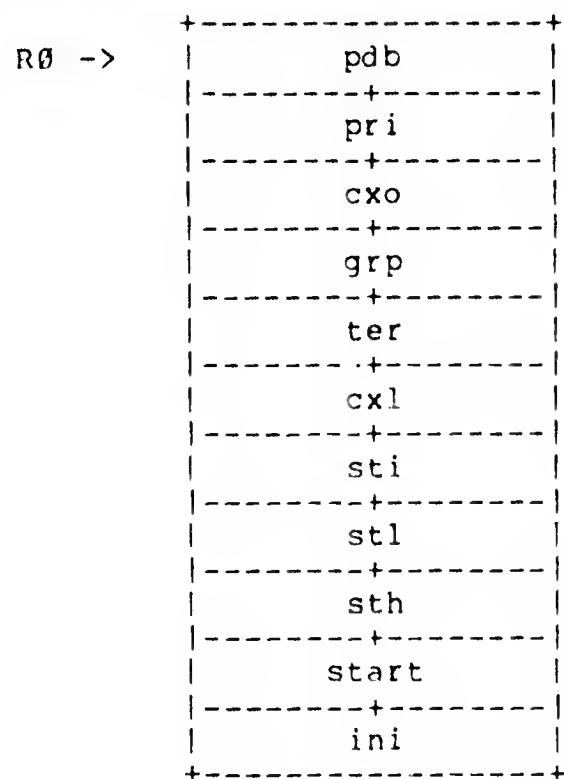
The first word of the passed PDB (the process index) must be zeroed.

The stack addresses sti, stl, and sth must be word addresses (even values).

The usable area of the process stack lies between stl and sth, exclusively. (The kernel uses the stl and sth locations for guard words.)

The size of the process stack, in bytes, must be greater than or equal to the value \$MINST, which the kernel defines to be the maximum number of bytes it uses on the process stack. In unmapped systems, the current value of \$MINST is 44(decimal) bytes; in mapped systems, the current value of \$MINST is 0. When calculating required stack space for a process, the user should add the process's own stack requirements to \$MINST.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the CRPC\$x macro has the following format:



## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
CRPCSS pdb=#P1,pri=#0,cxo=#0,grp=#6,ter=#END,cxl=#0,stl=#HIS,  
stl=#LOS,sth=#HIS,start=#BEGIN,ini=#0
```

### Semantics

The CRPCS primitive allocates a PCB representing the requested process in system-common memory and initializes it with the following:

- Attributes and values specified in the call
- The name, if any, contained in the PDB
- Attributes inherited from the parent process -- address space, mapping type, exception entry address, and exception bit mask, if any

The primitive then starts the new process by placing its PCB in the kernel's ready-active queue, from which it is scheduled for execution.

On successful return from the primitive call, the PDB passed by the caller contains information that can be used subsequently by other primitives for efficient access to the process. (See Section 3.1.6.)

The assigned priority value (pri) affects the scheduling of the process relative to all other processes.

The group code (grp) declares the process as a member of a particular exception-handling group. Some handler for this group will be invoked to manage exceptions caused by the process. (See the CCNDS primitive for further information. See the SERAS primitive concerning exception handling within the process.)

The termination entry point (ter) is the location to which control is transferred by the kernel in the event of an exception-caused abort or a Stop Process operation executed on the subject process. This allows the subject process to execute a "graceful termination" procedure, which must end with a Delete Process (DLPCS) request.

The CX\$FPP option (cxo) allows a general process using the floating-point processor to have the contents of those registers saved when it is context-switched, at the cost of additional PCB size and context switch time.

The CX\$MCX option, cxl argument, and ini argument collectively allow the process to add a single memory location to its switched context. (This feature is required by the MicroPower/Pascal compiler.)

The CX\$KT option causes the mapping registers to be saved during context switching, allowing a process with privileged, driver, or device-access mapping to modify its mapping. (See the DFSPCS macro.) This option is meaningless in an unmapped system; it incurs needless overhead if applied to a general process in a mapped environment.

The stl and sth values are used by the kernel for dynamic stack checking. Violation of either the lower or the upper boundary will cause a runtime exception to occur.

### Error Returns

E.BUSY - Insufficient memory; could not allocate a PCB.

E.IIID - Process with specified ID already exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.8 CRSTS (CREATE STRUCTURE)

Pascal equivalents: CREATE\_BINARY\_SEMAPHORE Function  
CREATE\_COUNTING\_SEMAPHORE Function  
CREATE\_QUEUE\_SEMAPHORE Function  
CREATE\_RING\_BUFFER Function

The Create Structure (CRSTS) primitive creates a specified, kernel-managed type of system data structure in system-common memory. The structure types -- binary and counting semaphores, queue semaphores, and ring buffers -- are defined by the kernel. All may be accessed directly only by the kernel.

If the structure is successfully created, the primitive returns the kernel-defined value TRUE in R0. If the structure cannot be created, due to lack of free system memory, the primitive returns the kernel-defined value FALSE in R0.

The CRSTS primitive permits a process to create named structures intended for interprocess synchronization and communication. These structures can be operated on in a controlled and reliable fashion through the use of other primitives.

#### Syntax

The three variants of the CRSTS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CRSTS	CRSTS [area,sdb,styp,satr,value]
CRSTSS	CRSTSS [sdb,styp,satr,value]
CRSTSP	CRSTSP [sdb,styp,satr,value]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of the user-constructed structure descriptor block (SDB) containing the name, if any, of the structure to be created and in which the kernel returns information identifying the structure. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=]sdb-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**styp**

The type of structure to be created, as indicated by predefined symbols. The structure-type symbols are:

ST.BSM	Binary semaphore
ST.CSM	Counting semaphore
ST.QSM	Queue semaphore
ST.RBF	Ring buffer
ST.UDF	Unformatted structure

These symbols are defined by the QUEDFS macro. This argument has the form:

[STYP=] type-symbol

**satr**

The ordering attributes, as indicated by predefined bit-mask symbols, for any structure and the access attributes for a ring buffer. The ordering attribute symbols are:

SASOFF FIFO ordering or:

SASOPR Priority ordering of:

1. The waiting process list of a binary, counting, or queue semaphore
2. The waiting output-process list of a ring buffer (processes waiting to get an element)

SASIFF FIFO ordering or:

SASIPR Priority ordering of:

1. The queue of packets in a queue semaphore
2. The waiting input-process list of a ring buffer (processes waiting to put an element)

The access attributes apply only to a ring buffer and affect the operation of the PELCS primitive (input access) and the GELCS primitive (output access). The access attributes are:

SASRIR Record-oriented (default) input access  
SASRIS Stream-oriented input access

SASROR Record-oriented (default) output access  
SASROS Stream-oriented output access

Note that SASIFF and SASIPR are not applicable to binary or counting semaphores. Two or more attribute symbols may be ORed as required for queue semaphores and ring buffers. These symbols are defined by the QUEDFS macro. This argument has the form:

[SATR=]attribute-symbol[!attribute-symbol]

## MACRO-11 PRIMITIVE SERVICE REQUESTS

value

Either the initial value of a semaphore variable or the buffer size for a ring buffer, depending on the type of structure requested. For a binary semaphore, value must be 0 or 1; for a counting semaphore, value may be a nonnegative integer; for a queue semaphore, value must be 0; for a ring buffer, value is the even number of bytes to be allocated for element space. For an undefined structure, value is the even number of bytes to be allocated. This argument has the form:

{VALUE=}integer-value

**Restrictions** - The first word of the passed SDR in the structure index -- must be zeroed.

The minimum size of a ring buffer is 8 bytes; the maximum, 8192 bytes. The number of bytes specified (value argument) must be even.

**Argument Block** - The calling argument block generated by CRSTSS must exist -- by the CRSTSx macro has the following format:

-----+-----+-----+-----+-----+-----+	-----+-----+-----+-----+-----+-----+				
R0 ->		sdb			
		-----+-----+-----+-----+-----+-----+			
		styp			
		-----+-----+-----+-----+-----+-----+			
		satr			
		-----+-----+-----+-----+-----+-----+			
		value			
		-----+-----+-----+-----+-----+-----+			

### Syntax Example

```
CRSTS - sdb=#SEM,styp=ST.RSM,satr=#SACCPR,value=$)
```

### Semantics

The CRSTS primitive allocates and initializes the requested structure in system-common memory and returns to the caller with the kernel-defined value TRUE in R0. If the structure is successfully created, it is named as specified in the passed SDR. On return, the SDR contains additional information that can subsequently be used by other primitives for optimized access to the structure.

If structure creation is unsuccessful due to insufficient space in the system free-memory pool, the primitive returns immediately to the caller with the kernel-defined value FALSE in R0.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/BASIC, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Error Returns

- E.IIID - Nonunique structure name; a similar structure already exists with the specified name.
- E.ILPM - An illegal structure type was specified.
- E.ILPR - Structure type used illegally; user code attempted to create a PCB.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.9 DAPK\$ (DEALLOCATE PACKET)

Pascal equivalent: DEALLOCATE\_PACKET Procedure

The Deallocate Packet (DAPK\$) primitive returns a message packet -- standard queue element -- to the kernel's free-packet pool. This primitive permits the caller to release a packet it has acquired via a Wait on Queue Semaphore (WAIQ\$ or WAQC\$) primitive operation when the packet is no longer needed.

The inverse of DAPK\$ is the ALPK\$ primitive. ALPK\$ allows a process to allocate -- obtain a pointer to -- a free packet for use with SGLQ\$ or SGQC\$.

#### Syntax

The three variants of the DAPK\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
---------	------------

DAPK\$	DAPK\$ [area,qelm]
--------	--------------------

DAPK\$S	DAPK\$S [qelm]
---------	----------------

DAPK\$P	DAPK\$P [qelm]
---------	----------------

Parameter	Definition
-----------	------------

area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:
------	--

[AREA=]arg-blk-address

qelm	The address of the packet that is to be deallocated. This argument has the form:
------	--

[QUELM=]packet-pointer

The argument must specify a word address in the kernel's data space.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the DAPK\$x macro has the following format:

R0 ->	+-----+   qelm   +-----+
-------	--------------------------------

#### Syntax Example

DAPK\$ area=#BLOCK,qelm=R3

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **Semantics**

The DAPKS primitive tests the pointer value to ensure that it lies within the kernel's data space. If it does, the primitive returns the assumed packet to the kernel's free-packet pool. If no other process is waiting for packet allocation, DAPKS returns control to the calling process.

If at least one process is waiting for packet allocation, the newly freed packet is allocated to the highest-priority waiting process, that process is unblocked, and the scheduler is called. This may cause the calling process to be preempted, depending on the priority of the unblocked process. (See the ALPK\$ primitive.)

### **Error Returns**

E.ADDR - Invalid address; the pointer value is not a word address or is not within the range of valid packet addresses.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.10 DEXCS (DISMISS EXCEPTION CONDITION)

Pascal equivalent: RELEASE\_EXCEPTION Procedure

The Dismiss Exception Condition (DEXCS) primitive is used by an exception-handler process to return an exception-wait process to the kernel for further disposition. The process changes to the appropriate ready state.

The primitive allows the handler to dispose of a PCB that it received on its exception queue, after processing the exception and determining a course of action. An action code specified in the DEXCS call directs the kernel to take one of three actions concerning the returned PCB: cancel the exception, abort the process, or pass the exception to the process's exception entry point, if any.

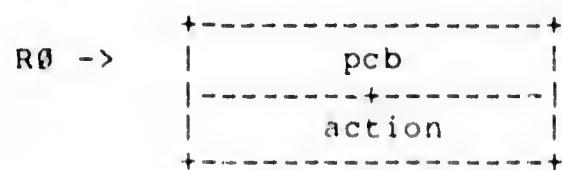
#### Syntax

The three variants of the DEXCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
DEXCS	DEXCS [area,pcb,action]
DEXCSS	DEXCSS [pcb,action]
DEXCSP	DEXCSP [pcb,action]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
pcb	The address of a pointer to the PCB that is to be returned to the kernel. This argument has the form:  [PCB=]pointer-address
action	The type of action, as indicated by a predefined action code symbol, to be taken by the kernel. The action code symbols are:  EA\$DIS      Dismiss exception; place process in ready state. EA\$ABT      Abort process by forcing termination entry point. EA\$PAS      Pass exception to the process, if possible; otherwise, abort.  These symbols are defined by the EXACTS macro. This argument has the form:  [ACTION=]action-symbol

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the DEXC\$X macro has the following format:



### Syntax Example

```
DEXCSS  pcb=R4,action=EA$DIS
```

### Semantics

The DEXC\$ primitive places the passed PCB, which was received in exception-wait-active state via the caller's exception queue, on the appropriate ready-state queue for disposition as requested in the call. (The PCB is placed on the ready-active queue unless it was suspended while in the exception wait state.) The kernel will take one of the following actions:

1. Cancel the exception, allowing the process to be reentered normally when it is rescheduled (action = EA\$DIS).
2. Abort the process, causing its termination entry to be forced when the process is rescheduled (action = EA\$ABT).
3. Pass the exception condition to the process's own exception entry point. If r such entry point exists, or if the process has not requested the particular type of exception, the termination entry point will be forced instead (action = EA\$PAS). See the SERAS primitive concerning exception handling within the process.

### Error Returns

E.ILPM - Illegal parameter specified -- an invalid PCB address or an illegal action code.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.11 DFSPCS (DEFINE STATIC PROCESS)

Pascal equivalent: Declaration of a [SYSTEM] PROGRAM

The Define Static Process (DFSPCS) assembly-time macro defines a program to be a static process -- the equivalent of a MicroPower/Pascal SYSTEM PROGRAM. A static process is permanently resident; it is installed during system initialization by the INIT routine, which is part of the MicroPower/Pascal runtime system. The installation of such a process requires an entry in the static process definition list, which drives INIT.

The DFSPCS macro is not a primitive call. Rather, it generates the required static process definition list entry for INIT during assembly. The complement of the DFSPCS macro for runtime creation of dynamic processes is CRPCS. See the description of processes in Chapter 2 for additional information.

A static process in a mapped environment has a specified mapping type: privileged (full system), driver (device handler), device access, or general. A dynamic process inherits its mapping type from its originating static process, since every dynamic process exists in the address space of a given static process. The characteristics of the mapping types are described in Section 2.1.5.

The PURE\$, PDAT\$, and IMPUR\$ program-sectioning macros should be used in conjunction with the DFSPCS macro to properly segregate read-only (ROM) and read/write (RAM only) program sections.

Note that the Define Static Process call is transparent to the Pascal user; no predefined MicroPower/Pascal procedure is equivalent to this macro. In Pascal, definition of a static process is implicit in the declaration of a program with the SYSTEM program attribute.

#### Syntax

The DFSPCS macro syntax is:

DFSPCS pid,pri,typ,cxo,grp,ter,cxl,sti,stl,sth,start,ini

Parameter	Definition
-----------	------------

pid	A 6-character ASCII string specifying the name of the static process. This argument must match the application program name (see the Restrictions section below). This argument has the form:
-----	---

[PID=]ascii-string

pri	The priority value (0 to 255) to be associated with the process. This argument has the form:
-----	--

[PRI=]integer-value

## MACRO-11 PRIMITIVE SERVICE REQUESTS

typ	The mapping type of the process, as indicated by predefined process-type symbols. The type symbols are PT.GEN for general process mapping, PT.PRV for privileged (full-system) process mapping, PT.DEV for device access process mapping, and PT.DRV for driver process mapping. These symbols are defined by the PTDFS macro. This argument has the form:  [TYP=]type-symbol
cxo	Any optional hardware context, as indicated by predefined bit-mask symbols, to be included (saved and restored) in the context switching performed for this process. The option symbols are CX\$FPP for the floating-point registers, CX\$KTR for the MMU registers, CX\$MCX for the single memory location specified by cxl, and CX\$STD for standard context switching only (or "save no additional context"). (CX\$MCX is intended for use primarily by the Pascal compiler.) The option symbols may be ORed as required. These symbols are defined by the CXODE\$ macro. This argument has the form:  [CXO=]option[!option]
grp	The exception-handling group to which the process belongs, as indicated by an arbitrary integer code value of 1 to 255. (See the Restrictions and Semantics sections below.) The exception group code values must be established by a convention agreed on by all process designers. This argument has the form:  [GRP=]integer-value
ter	The entry point of the termination instruction sequence for the process. (See the Semantics section below.) This argument has the form:  [TER=]instruction-address-value
cxl	The address of the user-memory location whose content is to be saved/restored when this process is context-switched. This argument is meaningful only if CX\$MCX is specified in cxo; otherwise, it must be 0. This argument has the form:  [CXL=]address-value
sti	The initial address of the user-allocated process stack. This argument has the form:  [STI=]top-of-stack-address
stl	The address of the low boundary of the process stack. This argument has the form:  [STL=]low-bound-address
sth	The address of the high boundary of the process stack. This argument has the form:  [STH=]high-bound-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**start**      The initial entry point for the process. This argument has the form:

[START=]first-instruction-address

**ini**      The initial value for the location cx1 -- the value to be stored in that location when the process is first switched into execution. This argument is meaningful only if CX\$MCX is specified in cx0; otherwise, it may be left null. This argument has the form:

[INI=]word-value

**Restrictions** - The static process name (pid) must match the application program name. The application program is named either at source level, by including a .TITLE statement at the beginning of the first application module that is to be linked by MERGE, or at build time, by using the RELOC /N switch.

Exception group code 0 may not be specified for a process, and codes 128 to 255 (200 to 377 octal) are reserved for possible future use.

The stack addresses sti, stl, and sth must be word addresses (even values).

The size of the process stack, in bytes, must be greater than or equal to the value \$INST, which the kernel defines to be the maximum number of bytes it uses on the process stack. In unmapped systems, the current value of \$MINST is 44(decimal) bytes; in mapped systems, the current value of \$MINST is 0. When calculating required stack space for a process, you should add the process's own stack requirements to \$MINST.

### Syntax Example

```
DFSPCS pid=MOVER,pri=5,typ=PT.PRV,cxo=CX$KT,grp=1,ter=ABT,cx1=0,  
sti=HIS,stl=LOS,sth=HIS,start=BEGIN,ini=0  
; Note: Only constants may be specified; do not use '#'.  
;
```

### Semantics

From the information specified in the call, the DFSPCS macro generates an entry for the static process definition list during assembly. This entry is used during system initialization, by the INIT procedure, to install the defined static process.

The process identifier (pid) can be used in primitive calls by other processes to refer to this process.

The assigned priority value (pri) affects the scheduling of the process relative to all other processes.

The process mapping type (typ) indicates the type of address mapping to be provided for this process in a mapped environment -- general, device access, driver, or privileged. The mapping types are described in detail in Section 2.1.7.

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

The CX\$FPP option (cxo) allows a general process using the floating-point processor to have the contents of those registers saved when it is context-switched, at the cost of additional PCB size and context switch time. The CX\$MCX option, cxl argument, and ini argument collectively allow the process to add a single memory location to its switched context. (This feature is required by the MicroPower/Pascal OTS.)

The CX\$KT option causes the mapping registers to be saved during context switching, allowing a process with privileged, device-access, or driver mapping to modify its mapping. This option is meaningless in an unmapped system; it incurs needless overhead if applied to a general process in a mapped environment.

The group code (grp) declares the process as a member of a particular exception-handling group. A handler for this group will be invoked to manage exceptions caused by the process. (See the CCND\$ primitive for further information. See the SERAS\$ primitive for exception handling within the process.)

The termination entry point (ter) is the location to which control is transferred by the kernel in the event of an exception-caused abort or a Stop Process operation executed on the subject process. This allows the subject process to execute a "graceful termination" procedure, which must end with a Delete Process (DLPC) request.

The kernel uses the stl and sth values for dynamic stack-limit checking. Violation of either the lower or upper boundary will cause a runtime exception to occur.

### **Error Returns**

Not applicable; this macro is not executable.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.12 DINT\$ (DISCONNECT FROM INTERRUPT)

Pascal equivalent: DISCONNECT\_INTERRUPT Procedure  
DISCONNECT\_SEMAPHORE Procedure

The Disconnect from Interrupt (DINT\$) primitive breaks the connection between a specified interrupt vector and the interrupt service routine (ISR) that it is connected to, if any. Further interrupts through that vector are ignored.

This primitive can be used only by the current owner of the vector. (The primitive will not ordinarily be used in a dedicated system environment, but is supplied for functional completeness.)

#### Syntax

The three variants of the DINT\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
DINT\$	DINT\$ [area,vec]
DINT\$S	DINT\$S [vec]
DINT\$P	DINT\$P [vec]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
vec	The address of the hardware interrupt vector to be disconnected from the ISR. This argument has the form:  [VEC=]vector-address

**Restrictions** - The specified vector, if connected, must have been connected by the calling process.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the DINT\$x macro has the following format:

```
R0 -> +-----+
          |       vec      |
          +-----+
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
DINT$ area=#argblk,vec=#300
```

### Semantics

The DINT\$ primitive reinitializes the interrupt dispatch block (IDB) associated with the specified vector to point to the null (do nothing) ISR. (The null ISR dismisses any interrupts from unconnected vectors after incrementing an unsolicited interrupt counter.)

If the specified vector is not connected at the time of the call, the primitive returns immediately to the caller, performing no operation.

### Error Returns

E.ILVC - Illegal vector specified (outside valid vector range V.MIN - V.MAX).

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **3.13 DLPC\$ (DELETE PROCESS)**

Pascal equivalent: None

The Delete Process (DLPC\$) primitive service deletes the calling process. This primitive permits a process, static or dynamic, to terminate itself. Delete Process is the only method of process termination.

Note that the Delete Process service is transparent to the Pascal user; there is no predefined MicroPower/Pascal procedure equivalent to the DLPC\$ macro. In Pascal, deletion of a process is implicit when a PROCESS or a SYSTEM program terminates -- when the final END statement of either entity is encountered or the END statement of a [TERMINATE] procedure is encountered.

#### **Syntax**

The DLPC\$ macro call syntax is:

DLPC\$

#### **Semantics**

The DLPC\$ primitive removes the caller's PCB from the run queue, deallocates the PCB -- returns it to the system free-memory pool -- removes the process name from the system name table, and calls the scheduler.

#### **Error Returns**

E.IIID - Nonunique process name.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.14 DLSTS (DELETE STRUCTURE)

Pascal equivalent: DESTROY Procedure

The Delete Structure (DLSTS) primitive deletes a specified semaphore or ring buffer from the system and deallocates the memory space associated with it. The structure must not be in use at the time of the call. That is, no processes may be blocked on the structure, a queue semaphore must have no packets on its queue, and a ring buffer must be empty.

This service permits a process to release the memory allocated to a structure that is no longer needed.

#### Syntax

The three variants of the DLSTS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
DLSTS	DLSTS [area,sdb]
DLST\$S	DLST\$S [sdb]
DLST\$P	DLST\$P [sdb]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form: [AREA=]arg-blk-address
sdb	The address of the structure descriptor block (SDB) that identifies the structure to be deleted. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form: [SDB=]sdb-address

**Restrictions** - The structure identified by the passed SDB must be a semaphore -- binary, counting, or queue -- or a ring buffer.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the DLSTSx macros has the following format:

```
    +-----+
R@ -> |      sdb      |
    +-----+
```

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **Syntax Example**

```
DLSTS$  sdb=$SEM
```

### **Semantics**

The DLSTS primitive checks the structure to ensure that it is not in use. If the structure is not in use, the primitive removes the structure name from the system name table, if named, returns the space allocated to the structure to the kernel's free-memory pool, and returns control to the caller. If the structure is in use, the primitive returns to the caller with a "busy" error indication.

### **Error Returns**

E.IIID - Either the structure identifier or the structure name is invalid.

E.BUSY - The structure is in use and cannot be deleted.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.15 FORKS (FORK PROCESSING)

Pascal equivalent: None

The Fork Processing (FORK\$) service request is used by an interrupt service routine (ISR) to discontinue execution at interrupt level and to resume as a fork routine at system level. The execution of a fork routine is deferred until all interrupts have been serviced and any interrupted primitive operation has been completed, but occurs before return to process level. The code following the FORKS call becomes the body of the fork routine, which must terminate via a RETURN, as does a normal ISR.

An ISR must issue a FORKS request before requesting any primitive service. This ensures that an ISR will not execute a primitive operation from interrupt level and thereby cause the kernel to be reentered. (Kernel primitive operations can be interrupted but not reentered.) Note that the kernel FORK service is not itself a primitive operation.

Fork routines have ISR context and a higher software priority than does any process but run at CPU priority 0. Each fork routine has a priority relative to other fork routines; this relative priority is specified in the FORKS request. Fork routines are scheduled and run from a special queue that is independent of process scheduling.

The FORK mechanism guarantees sequential execution of primitives while permitting their use within ISRs by serializing execution of ISR segments that contain primitive requests.

#### Syntax

The syntax of the FORKS macro call is:

```
FORK$ pri
```

Parameter	Definition
-----------	------------

pri	The address of a word containing the software priority value to be associated with the request. The priority value can range from 0 to 65535. This parameter has the form:
-----	--

```
[PRI=]word-address
```

When the request is issued, R5 must point to the fork block for the ISR; see the Semantics section below. (On normal entry to an ISR, R5 points to the fork block contained in the interrupt dispatch block associated with the ISR.)

**Restrictions** - This service may be requested only within an ISR executing at less than priority level 7.

Before issuing a FORKS request, the ISR must purge the stack of any data it has pushed.

A priority-7 ISR must issue a P7SYSS request before issuing a FORKS request.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
FORKS #5 ; Note: FORKS 5 would be invalid!
```

### Semantics

When an ISR requests fork-level processing, abbreviated ISR context (R3, R4, and PC) is saved in the fork block pointed to by R5, and the fork block is placed on the fork queue. The interrupt is then dismissed, by a RETURN to the interrupt dispatcher, which allows any pending interrupts to occur and processing of any lower-level interrupted ISR to continue. If a primitive operation was interrupted, it is allowed to complete. The kernel assumes that nothing has been left on the stack between the ISR entry and a FORKS request.

Before returning to process level, the kernel processes the fork request queue. The fork blocks are individually dequeued and the corresponding fork routines executed at CPU priority 0. Each fork routine must purge the stack, if used, and terminate with a RETURN instruction -- the normal ISR exit procedure.

The fork queue is priority-ordered; the scheduling priority associated with a fork routine is specified in the request. The specified priority can range from 0 to 65535 and bears no relationship to the CPU (hardware) priority at which the ISR normally executes; nor does it bear any relationship to process priorities.

Registers R0-R5 are available for use after a SFORK request.

### Error Returns

If the FORKS request returns with the carry (C) bit set, the FORKS was unsuccessful because an already pending FORKS request for that ISR had not been started yet. (Two FORKS requests -- at most -- can be pending, provided that the first FORKS routine has started execution.)

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.16 GE' , (CONDITIONAL GET ELEMENT)

Pascal equivalent: COND\_GET\_ELEMENT Function

The Conditional Get Element (GELCS) primitive implements a nonblocking form of Get Element operation; compare with the unconditional GELMS primitive. GELCS attempts to extract the requested number of bytes from the ring buffer but does not block the calling process if the request cannot be satisfied. The output-access mode of the ring buffer, record or stream, determines how the primitive attempts to satisfy the Get request: whether by a full or a partial transfer, as described below. In either case, however, GELCS returns to the caller with a value in R0 indicating how many bytes are still needed to fully satisfy the request. A return value of 0 indicates that the request has been fully satisfied; all the bytes specified in the call have been successfully transferred from the ring buffer.

The output-access mode of a ring buffer is specified as either record-oriented or stream-oriented when the structure is created; see the CRSTS primitive. For a ring buffer with record-mode output, the default, GELCS attempts to satisfy the request with a full transfer only. If the ring buffer does not contain as many bytes as requested, the primitive returns immediately to the caller, with a value equal to the number of bytes specified in the request, indicating that no bytes were obtained.

For a ring buffer with stream-mode output, GELCS attempts to satisfy the request with either a full or a partial transfer. That is, the primitive gets as many bytes as are available in the ring buffer, up to the number requested, and returns a value indicating the number that remains to be obtained, if any.

Note the implication that if another process is blocked on the ring buffer, waiting for its GELMS request to be satisfied, no bytes are available.

The complementary PELMS and PELCS primitives allow a process to put bytes into a ring buffer.

#### Syntax

The three variants of the GELCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
GELCS	GELCS [area,sdb,bufptr,bufcnt]
GELCSS	GELCSS [sdb,bufptr,bufcnt]
GELCSP	GELCSP [sdb,bufptr,bufcnt]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed or found, if already existent. This argument has the form: [AREA=]arg-blk-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**sdb**      The address of a structure descriptor block (SDB) that identifies the ring buffer from which bytes are to be extracted. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=] sdb-address

**bufptr**      The address of the user's buffer to receive the data element. This argument has the form:

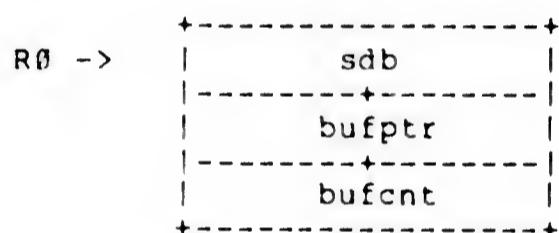
[BUFPTR=] buffer-address

**bufcnt**      The number of bytes to be transferred to the buffer pointed to by bufptr. This argument has the form:

[BUFCNT=] integer

**Restrictions** - If the ring buffer's output-access mode is record, the number of bytes requested must not exceed the size of the ring buffer. (If it does, the Get request will never be satisfied.)

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the GELC\$ macros has the following format:



### Syntax Example

```
GELCSS sdb=#TSTRING,bufptr=#LOW,bufcnt=#10.
```

### Semantics

If the specified ring buffer's output-access attribute is SASROR (record mode), the GELCS primitive tests the ring buffer for bufcnt bytes of available data. If at least that amount of data is available, the primitive copies bufcnt bytes from the ring buffer to the caller's buffer, deletes the corresponding bytes from the ring buffer, and returns control to the caller, with 0 in R0. If the ring buffer contains less than bufcnt bytes, GELCS returns immediately with the value bufcnt in R0, indicating that no bytes were obtained.

If the specified ring buffer's output-access attribute is SASROS (stream mode), GELCS gets as many bytes as are available in the ring buffer, up to the number requested, and returns control to the caller with the value (bufcnt minus bytes transferred) in R0.

### Error Returns

E.IID - Invalid structure description (index or name); no such ring buffer exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.17 GELMS (GET ELEMENT)

Pascal equivalent: GET\_ELEMENT Procedure

The Get Element (GELMS) primitive extracts a specified number of data bytes from a ring buffer and transfers them to the caller's buffer area. If too few bytes are in the ring buffer to fully satisfy the Get request, the calling process blocks on the ring buffer, waiting for more bytes to become available. (The data transfer may occur in increments while the process is blocked on the ring buffer.)

If two or more processes are getting data from the same ring buffer, the calling process will also block if another process is already waiting for its Get request to be satisfied. In this case, the calling process must wait its turn for access to the buffer, since sequential access to a ring buffer is ensured among multiple readers, as well as among multiple writers. Thus, the process heading the waiting Get Process list is never displaced by a higher-priority process.

The complementary PELMS and PELCS primitives allow a process to put elements into a ring buffer.

The conditional, nonblocking form of GELMS is the GELCS primitive.

#### Syntax

The three variants of the GELMS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
GELMS	GELMS [area,sdb,bufptr,bufcnt]
GELMSS	GELMSS [sdb,bufptr,bufcnt]
GELMSP	GELMSP [sdb,bufptr,bufcnt]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of a structure descriptor block (SDB) that identifies the ring buffer from which bytes are to be extracted. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=] sdb-address
bufptr	The address of the user's buffer that is to receive the ring buffer element. This argument has the form:  [BUFPTR=]buffer-address

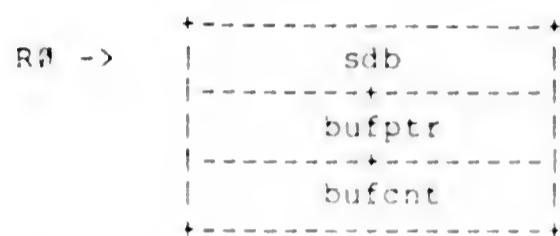
## MACRO-11 PRIMITIVE SERVICE REQUESTS

**bufcnt**      The number of bytes to be transferred to the buffer pointed to by bufptr. This argument has the form:

[BUFCNT=]integer

**Restrictions** - If the ring buffer's output-access mode is record, the number of bytes requested must not exceed the size of the ring buffer. (If it does, the Get request will never be satisfied.)

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the GELMSx macros has the following format:



### Syntax Example

```
GELMS area=$ARGBLK,bufptr=$BUFL,bufcr=$SIZE
```

### Semantics

If no other process is waiting to get an element from the specified ring buffer, the GELMS primitive tests the buffer for bufcnt bytes of available data. If at least that amount of data is available, the primitive copies the requested number of bytes from the ring buffer to the caller's buffer, deletes the corresponding bytes from the ring buffer, and returns control to the caller.

If the ring buffer contains less than bufcnt bytes, the primitive places the calling process at the head of the ring buffer's list of waiting Get processes, transfers any available bytes, and calls the scheduler. When enough additional bytes become available as a result of one or more subsequent Put Element operations, the transfer is completed, and the waiting process is unblocked.

If another process is waiting to get data from the ring buffer at the time of the call, implying active read access, the calling process is placed on the buffer's waiting Get Process list at a position below the head of the list. Processes are queued on the waiting Get Process list in either FIFO or priority order, depending on the output-ordering attribute of the ring buffer (see CRSTS). However, the process at the head of the list -- the process with active read access -- is never displaced by another process. The head-of-the-list position changes only when the process in that position is deleted from the waiting process list as a consequence of being either unblocked or stopped by another process.

### Error Returns

E.IIID - Invalid structure description (index or name); no such ring buffer exists.

E.ILPM - The bufcnt value exceeds the size of the ring buffer for a record-mode operation.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.18 GTSTS (GET PROCESS STATUS)

Pascal equivalent: GET\_STATUS Procedure

The Get Process Status (GTSTS) primitive returns information about the status of a process when the primitive service is invoked. The information includes the priority, type, and group code of the process, which does not change, as well as the state code, suspend count, and index of the semaphore or ring buffer on which the process is blocked, if any. GTSTS returns the information in a user-specified buffer area.

Note that process status information is dynamic and may be invalid by the time it is available to the caller (except for priority, type, and group).

#### Syntax

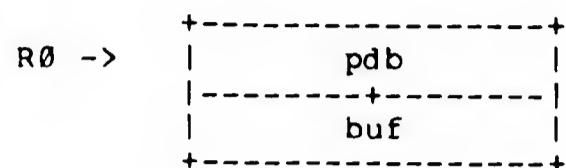
The three variants of the GTSTS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
GTSTS	GTSTS [area,pdb,buf]
GTSTSS	GTSTSS [pdb,buf]
GTSTSP	GTSTSP [pdb,buf]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [ARPA=]arg-blk-address
pdb	The address of the process descriptor block (PDB) that identifies the process to be reported on or @. If @ is specified, the calling process is implied. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:  [PDB=]pdः-address
buf	The address of a 5-word user-memory area in which the status information is to be returned by the primitive. This argument has the form:  [BUFA=]buffer-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block -** The calling argument block generated or assumed to exist by the GTST\$X macros has the following format:



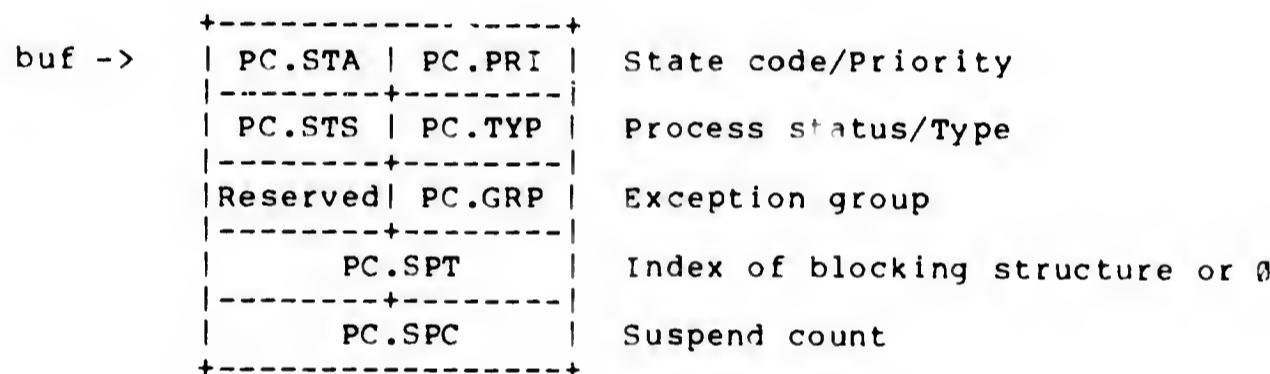
### Syntax Example

```
GTST$S  pdb=#0,buf=#MYBUF
```

### Semantics

The GTST\$ primitive copies five words of status information from the PCB of the specified process to the caller's buffer area and returns control to the caller.

The information is returned in the user's buffer in the following form:



### Error Returns

E.IIID - Illegal process identifier or name; the passed PDB does not validly describe a process.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.19 GVAL\$ (RETURN STRUCTURE VALUE)

Pascal equivalent: GET\_VALUE Procedure

The Return Structure Value (GVAL\$) primitive provides type and value information about a specified semaphore or ring buffer. GVAL\$ returns a code indicating the structure type -- binary, counting, or queue semaphore or ring buffer -- and returns a value whose meaning is dependent on the structure type. For example, the signal count is returned for a counting semaphore, and the element count is returned for a ring buffer.

Note that the value of a structure may change immediately after it is inspected. Therefore, the information provided by this primitive must be used with caution in order to prevent the introduction of race conditions.

#### Syntax

The three variants of the GVAL\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

##### Variant

##### Macro Call

GVAL\$

GVAL\$ [area,sdb,type,val]

GVALSS

GVALSS [sdb,type,val]

GVALSP

GVALSP [sdb,type,val]

##### Parameter

##### Definition

area

The address of a user-memory area in which the calling argument block is to be constructed or found, if already existent. This argument has the form:

[AREA=]arg-blk-address

sdb

The address of the structure descriptor block (SDB) that identifies the structure to be inspected. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=]sdb-address

type

The address of the location in which the type code is to be returned by the primitive. This argument has the form:

[TYPE=]word-address

val

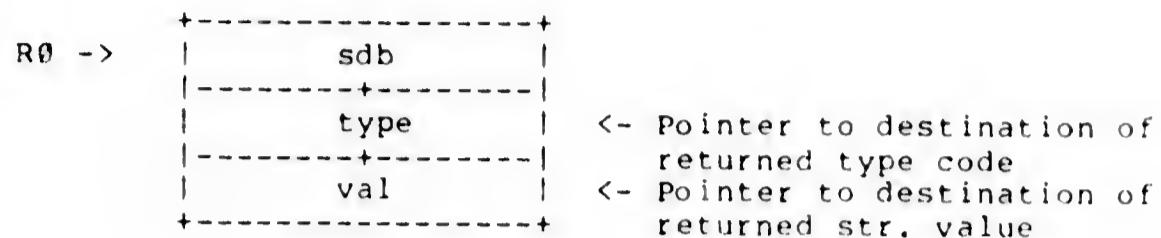
The address of the location in which the structure value is to be returned by the primitive. This argument has the form:

[VAL=]word-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Restrictions** - The structure identified by the passed SDB must be a semaphore -- binary, counting, or queue -- or a ring buffer.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the GVAL\$ macros has the following format:



### Syntax Example

```
GVAL$  sdb=#BSEM,type=#BTYPE,val=#BVAL
```

### Semantics

The GVAL\$ primitive inspects the type and value of the specified structure, stores the type code and structure value in the user location (type and val) pointed to by the calling argument block, and returns control to the call site.

The returned type code corresponds to one of the following structure-type symbols defined by the QUEDFS macro:

ST.BSM	Binary semaphore
ST.CSM	Counting semaphore
ST.QSM	Queue semaphore
ST.RBF	Ring buffer

The significance of the returned structure value varies according to structure type, as follows:

1. For ST.BSM, the value of the gate variable (0 or 1)
2. For ST.CSM, the count of pending signals (0 or positive)
3. For ST.QSM, the count of queued packets (0 or positive)
4. For ST.RBF, the count of data bytes in the ring buffer

### Error Returns

E.IID - Invalid structure identifier or structure name; not a semaphore or ring buffer.

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **3.20 IMPUR\$ (DEFINE AN IMPURE PROGRAM DATA SECTION)**

Pascal equivalent: None

The IMPUR\$ macro declares a program section of impure data within a MicroPower/Pascal process. A program section declared with IMPUR\$ will be given the read/write attribute and must be placed in RAM.

The IMPUR\$ macro is not a primitive call. Rather, it is an assembly-time macro that is used in conjunction with the DFSPCS, PURE\$, and PDAT\$ macros to properly segregate read-only (ROM-able) and read/write (RAM-only) program sections. During the system build process, program sections that have been declared with the IMPUR\$, PURE\$, and PDAT\$ macros are grouped according to the read-only versus read/write attribute by the RELOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.)

Note that no predefined MicroPower/Pascal procedure is equivalent to the IMPUR\$ macro. Program sectioning is provided transparently by the MicroPower/Pascal compiler.

#### **Syntax**

The syntax of the IMPUR\$ macro call is:

**IMPUR\$**

#### **Semantics**

At assembly time, the IMPUR\$ macro generates a p-sect definition with the default attributes RW (read/write), LCL (local), and D (data).

#### **Error Returns**

Not applicable; this macro is not executable.

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **3.21 PDATS (DEFINE A PURE PROGRAM DATA SECTION)**

Pascal equivalent: None

The PDATS macro declares a program section of pure data within a MicroPower/Pascal process. A program section declared with PDATS will be given the read-only attribute and may be placed in ROM.

The PDATS macro is not a primitive call. Rather, it is an assembly-time macro that is used in conjunction with the DEFPDS, PURES, and IMPURS macros to properly segregate read-only (ROM-able) and read/write (RAM-only) program sections. During the system build process, program sections that have been declared with the PDATS, PURES, and IMPURS macros are grouped according to the read-only versus read/write attribute by the PFLOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.)

Note that no predefined MicroPower/Pascal procedure is equivalent to the PDATS macro. Program sectioning is provided transparently by the MicroPower/Pascal compiler.

#### **Syntax**

The syntax of the PDATS macro call is:

PDATS

#### **Semantics**

At assembly time, the PDATS macro generates a p-sect definition with the default attributes RO (read-only), LCL (local), and D (data).

#### **Error Returns**

Not applicable; this macro is not executable.

## MACRO-II PRIMITIVE SERVICE REQUESTS

### 3.22 PELCS (CONDITIONAL PUT ELEMENT)

Pascal equivalent: COND\_PUT\_ELEMENT Function

The Conditional Put Element (PELCS) primitive implements a nonblocking form of Put Element operation; compare with the unconditional PELMS primitive. PELCS attempts to copy the requested number of data bytes from the caller's buffer area to a ring buffer but does not block the calling process if the request cannot be satisfied. The input-access mode of the ring buffer, record or stream, determines whether the primitive attempts to satisfy the put request by a full or by a partial transfer, as described below. In either case, however, PELCS returns to the caller with a value in R0 indicating how many bytes remain to be transferred. A return value of 0 indicates that the request has been fully satisfied -- all the bytes specified in the call have been successfully put to the ring buffer.

The input-access mode of a ring buffer is specified as either record-oriented or stream-oriented when the structure is created; see the CRSTS primitive. For a ring buffer with record-mode input, the default, PELCS attempts to satisfy the request with a full transfer only. If the ring buffer has insufficient space for all the bytes to be put, the primitive returns immediately to the caller, with a value equal to the number of bytes specified in the request, indicating that no bytes were copied.

For a ring buffer with stream-mode input, PELCS attempts to satisfy the request with either a full or a partial transfer. That is, the primitive puts all the bytes that can currently be accommodated in the buffer -- none, some, or all those requested -- and returns a value indicating the remainder, if any.

Note the implication that if another process is blocked on the ring buffer, waiting for its PELMS request to be satisfied, no space is available.

The complementary GELMS and GELCS primitives allow a process to extract an element from a ring buffer, freeing the corresponding space.

#### Syntax

The three variants of the PELCS macro and their respective macro calls are listed below. (The differences are described in Section 3.2.1.)

Variant	Macro Call
PELCS	PELCS (area,sdb,bufptr,bufcnt)
PELCSS	PELCSS (sdb,bufptr,bufcnt)
PELCSP	PELCSP (sdb,bufptr,bufcnt)

#### Parameter      Definition

area      The address of a user-memory area in which the calling argument block is to be constructed if found if already existent. This argument has the form:

'ARFA=1ata-blk-address'

## MACRO-11 PRIMITIVE SERVICE REQUESTS

sdb      The address of a structure descriptor block (SDB) that identifies the ring buffer to which bytes are to be transferred. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

    [SDB=]sdb-address

bufptr      The address of the user's buffer containing bytes to be transferred to the ring buffer. This argument has the form:

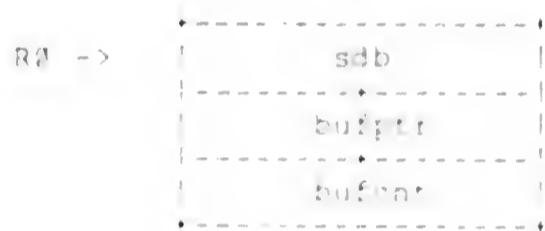
    [BUFFPTR=]buffer-address

bufcnt      The number of bytes to be transferred. This argument has the form:

    [BUFCONT=]integer

**Restrictions** - If the ring buffer's input-access mode is record, the number of bytes requested must not exceed the size of the ring buffer.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the PELCSx macros has the following format:



### Syntax Example

PELCS ntyp=\$ARCBLK,bufptr=\$MSG,bufcnt=\$MSGLEN

### Semantics

If the specified ring buffer's input-access attribute is SACRER (record mode), the PELCS primitive tests the ring buffer for bufcnt bytes of available space. If at least that amount of space is available, the primitive copies bufcnt bytes of data from the caller's buffer to the ring buffer and returns control to the caller, with R0 in RA. If less than bufcnt bytes of space is available, PELCS returns immediately with the value bufcnt in RA, indicating that no bytes were put.

If the specified ring buffer's input-access attribute is SACRST (stream mode), PELCS copies as many bytes from the caller's buffer as can be accommodated in the ring buffer and returns control to the caller, with the value (bufcnt minus bytes copied) in RA.

### Error Returns

F.IID - invalid structure descriptor (index or name); no such ring buffer exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.23 PELMS (PUT ELEMENT)

Pascal equivalent: PUT\_ELEMENT Procedure

The Put Element (PELMS) primitive copies a specified number of data bytes from the caller's buffer area to a ring buffer. If the ring buffer has insufficient space for the number of bytes to be put, the calling process blocks on the ring buffer until enough space becomes available. (The data transfer may occur in increments while the process is blocked on the ring buffer.)

If two or more processes are putting data into the same ring buffer, the calling process will also block if another process is already waiting for its Put request to be satisfied. In this case, the calling process must wait its turn for access to the buffer, since sequential access to a ring buffer is ensured among multiple writers, as well as among multiple readers. Thus, the process heading the waiting Put process list is never displaced by a higher-priority process.

The complementary GELMS primitive allows a process to extract an element from a ring buffer, freeing the corresponding space.

The conditional, nonblocking form of PELMS is the PELCS primitive.

#### Syntax

The three variants of the PELMS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

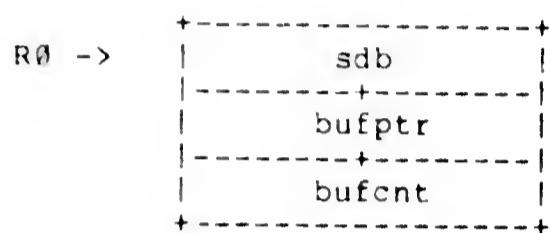
Variant	Macro Call
PELMS	PELMS [area,sdb,bufptr,bufcnt]
PELMSS	PELMSS [sdb,bufptr,bufcnt]
PELMSP	PELMSP [sdb,bufptr,bufcnt]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form: [AREA=]arg-blk-address
sdb	The address of a structure descriptor block (SDB) that identifies the ring buffer to which bytes are to be transferred. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form: [SDB=]sdb-address
bufptr	The address of the user's buffer containing bytes to be transferred to the ring buffer. This argument has the form: [BUFPTR=]buffer-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

bufcnt      The number of bytes to be transferred. This argument has the form:  
[BUFCNT=] integer

**Restrictions** - If the ring buffer's input-access mode is record, the number of bytes requested must not exceed the size of the ring buffer.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the PELMSx macros has the following format:



### Syntax Example

```
PELMSS sdb=#OUTRNG,bufptr=#INFO,bufcnt=#INFLEN
```

### Semantics

If no other process is waiting to put data into the specified ring buffer, the PELMS primitive tests the buffer for bufcnt bytes of available space. If at least that amount of space is available, the primitive copies the data from the caller's buffer to the ring buffer and returns control to the caller.

If there is insufficient space for the entire transfer, the primitive places the calling process at the head of the ring buffer's list of waiting Put processes, copies the bytes that can be accommodated, if any, and calls the scheduler. When enough additional space becomes available as a result of one or more subsequent Get Element operations, the transfer is completed, and the waiting process is unblocked.

If another process is waiting to put data into the ring buffer at the time of the call, implying active write access, the calling process is placed on the buffer's waiting Put process list at a position below the head of the list. Processes are queued on the waiting process list in either FIFO or priority order, depending on the input-ordering attribute of the ring buffer (see CRSTS). However, the process at the head of the list -- the process with active write access -- is never displaced by another process. The head-of-the-list position changes only when the process in that position is deleted from the waiting process list as a consequence of being either unblocked or stopped by another process.

### Error Returns

- E.IIID - Invalid structure description (index or name); no such ring buffer exists.
- E.ILPM - The bufcnt value exceeds the size of the ring buffer for a record-mode operation.

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **3.24 PURE\$ (DEFINE A PURE PROGRAM INSTRUCTION SECTION)**

Pascal equivalent: None

The PURE\$ macro declares a program section of pure code within a MicroPower/Pascal process. A program section declared with PURE\$ will be given the read-only attribute and may be placed in ROM. (Read-only program sections for data are declared by the PDATS macro, described elsewhere in this chapter.)

The PURE\$ macro is not a primitive call. Rather, it is an assembly-time macro that is used in conjunction with the DFSPCS\$, PDATS\$, and IMPUR\$ macros to properly segregate read-only (ROM-able) and read/write (RAM-only) program sections. During the system build process, program sections that have been declared with the PURE\$, IMPUR\$, and PDATS macros are grouped according to the read-only versus read/write attribute by the RELOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.)

Note that no predefined MicroPower/Pascal procedure is equivalent to the PURE\$ macro. Program sectioning is provided transparently by the MicroPower/Pascal compiler.

#### **Syntax**

The syntax of the PURE\$ macro call is:

**PURE\$**

#### **Semantics**

At assembly time, the PURE\$ macro generates a p-sect definition with the default attributes RO (read-only), LCL (local), and I (instruction).

#### **Error Returns**

Not applicable; this macro is not executable.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.25 P7SYSS (DROP CPU PRIORITY)

Pascal equivalent: None

The Drop CPU Priority (P7SYSS) service request allows an interrupt service routine (ISR) running at hardware priority 7 to enter system state -- to gain normal ISR context -- and to lower the CPU priority to a specified level. A priority-7 ISR must enter system state before lowering CPU priority and before issuing a FORKS request.

Note that the P7SYSS service is not a primitive operation. Rather, it is part of the interrupt dispatcher.

#### Syntax

The syntax of the P7SYSS macro call is:

P7SYSS pri

Parameter	Definition
pri	The desired processor priority level -- an integer of 0 to 6. This parameter has the form: [PRI=]integer

**Restrictions** - This service may be requested only within an ISR that is initially dispatched to at processor priority 7.

The contents of all registers must be the same as they were on entry to the ISR when the P7SYSS request is executed.

#### Syntax Example

P7SYSS 4 ; Note: P7SYSS #4 would be invalid!

#### Semantics

On return from the P7SYSS request, at the instruction following the call, the ISR is executing with normal ISR context at the specified hardware priority.

#### Error Returns

None.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.26 RBUFS (RESET RING BUFFER)

Pascal equivalent: RESET\_RING\_BUFFER Procedure

The Reset Ring Buffer (RBUFS) primitive resets the specified ring buffer by emptying it of data. This allows a process to cancel an I/O sequence and to flush the associated ring buffer without issuing multiple GELMS requests.

The RBUFS request is analogous to a GELMS request in that the caller is treated as a getting process for purposes of synchronization. That is, if any other process is blocked on the ring buffer, waiting for a GELMS request to be satisfied, the calling process is blocked and must wait its turn for read access to the buffer, just as it would for a GELMS request.

Note also that the RBUFS request does not inhibit any concurrent attempt by another process to put an element into the buffer. Thus, in certain applications the ring buffer may not be empty by the time control returns to the caller.

#### Syntax

The three variants of the RBUFS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
RBUFS	RBUFS [area,sdb]
RBUFSS	RBUFSS [sdb]
RBUFSP	RBUFSP [sdb]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  (AREA=)arg-blk-address
sdb	The address of the structure descriptor block (SDB) that identifies the ring buffer to be reset. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  (SDB=)sdb-address

**Argument Block** - The calling argument block generate -- or assumed to exist -- by the RBUFSx macros has the following format:



## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
RBUFS SDB=$TTRING
```

### Semantics

If no other process is waiting to get bytes from the specified ring buffer, the RBUFS primitive deletes any available bytes from the buffer and returns control to the caller.

If another process is waiting to get bytes from the ring buffer, RBUFS places the calling process on the buffer's waiting Get Process list at a point below the head of the list, as described for a GBLMS request, and calls the scheduler. When the blocked process gains read access to the ring buffer, the buffer is emptied, and the process is unblocked.

### Error Returns

E.IID - Invalid structure description (index or name); no such ring buffer exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.27 RCVCS (CONDITIONAL RECEIVE DATA)

Pascal Equivalent: COND\_RECVIVE Function  
COND\_RECVIVE\_AB Function

The Conditional Receive Data (RCVCS) primitive implements the nonblocking form of the RCVNS operation. RCVCS tests a specified queue semaphore for an available packet. If one is available, RCVCS obtains the packet and copies data from or through it to the caller's buffer space. If no packet is available, however, the primitive returns control immediately to the caller, instead of blocking the process on the semaphore, as is done by RCVNS. If the receive operation was performed, the primitive returns the kernel-defined value TRUE in R0. If it was not, the kernel-defined value FALSE is returned in R0.

This primitive is the analog of WAQS for use by general and device-access processes, which cannot access a packet directly in a mapped environment. RCVCS permits any type of process to conditionally receive data from another process through a packet. The complementary SENDS and SNDS primitives permit any type of process to send data through a packet. (See the SCQS primitive for further information about packets.)

The message-reception features of RCVCS are identical to those provided by RCVNS -- the copying of messages sent either by value or by reference. The only functional difference between the two primitives is the unconditional wait performed by SENDS versus the conditional wait performed by RCVCS.

#### Syntax

The three variants of the RCVCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
RCVCS	RCVCS (area, sdb, rtnptr, vlen, vbuf, rlen, rbuf)
RCVCSS	RCVCSS (sdb, rtnptr, vlen, vbuf, rlen, rbuf)
RCVCSP	RCVCSP (sdb, rtnptr, vlen, vbuf, rlen, rbuf)

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  TAREA:TA:m:h:k:d:ress

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**sdb** The address of the structure descriptor block (SDB) that identifies the queue semaphore to be tested for an available packet. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=] sdb-address

**rtnptr** The address of a 4-word area in which information about the Receive operation is to be returned by the primitive. (This area is not modified if the Receive operation is unsuccessful, that is, if R0 contains 0 on return from the primitive.) The format of the information returned is shown below. This argument has the form:

[RTNPTR=] word-address

**vlen** The length, in bytes, of the buffer pointed to by vbuf. This argument limits the amount of by-value data (if any) to be copied from the packet. The value of this argument can range from 0 to 34. This argument has the form:

[VLEN=] integer

**vbuf** The address of the buffer area in which data sent by value is to be copied. This argument has the form:

[VBUF=] address

This argument is significant only if vlen is nonzero.

**rlen** The length, in bytes, of the buffer pointed to by rbuf. This argument limits the amount of by-reference data, if any, to be copied from the sender's buffer. If this argument is 0 and if a message reference exists in the packet, the message is not copied; the reference itself is returned in the area pointed to by rtnptr, however. This argument has the form:

[RLEN=] integer

**rbuf** The address of the buffer area in which data sent by reference is to be copied. This argument has the form:

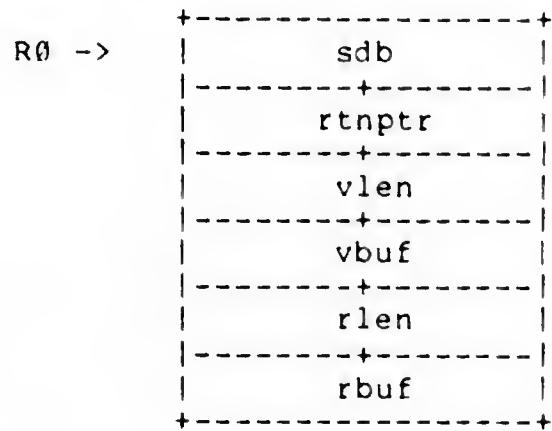
[RBUF=] address

This argument is significant only if rlen is nonzero.

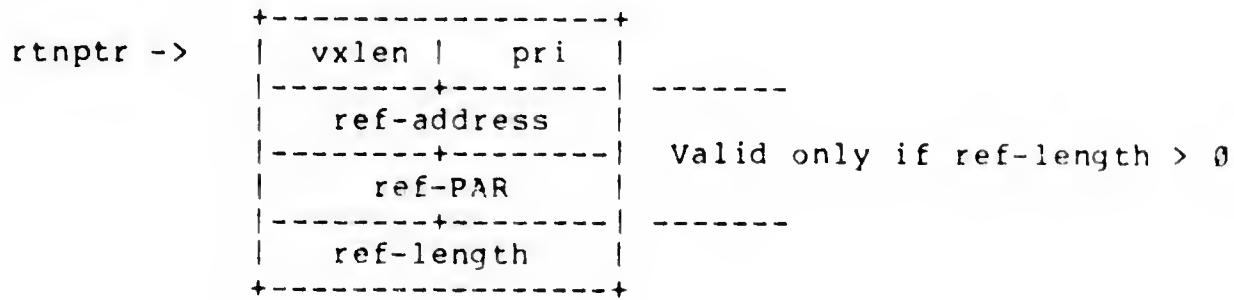
**Restrictions** - The value of parameter rlen may not exceed 8129.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the RCVC\$*x* macros has the following format:



**Format of Information Returned** - The information returned to the caller in the 4-word area pointed to by *rtnptr* is in the following form:



Parameter	Definition
pri	The priority value that was assigned to the packet by the Send operation.
vxlen	The number of bytes sent by value. (Note that this value may be greater than the number of bytes received, which is limited by the vlen argument.) If 0, no data was sent by value.
ref-address	The address of the sender's by-reference buffer, if any. This return value is valid only if the ref-length return value is nonzero; otherwise, the contents of this word are unpredictable.
ref-PAR	The value of the page address register that maps the sender's by-reference buffer, if any. This return value is valid only if the ref-length return value is nonzero; otherwise, the contents of this word are unpredictable.
ref-length	The number of bytes sent by reference. (Note that this value may be greater than the number of bytes received, which is limited by the rlen argument.) If 0, no data was sent by reference.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
RCVCS area=#ARGBLK,sdb=#QSEM,rtnptr=#INFO,vlen=#20.,  
vbuf=#DATA,rlen=#0,rbuf=#0
```

### Semantics

The RCVCS primitive tests the specified queue semaphore for an available packet. If a packet is available, the primitive removes it from the semaphore's packet queue and then performs the following actions, as governed by the arguments specified in the call:

1. Copies data sent by value, if any, from the packet in kernel space to the caller's vbuf area. The number of bytes copied is the lesser of the vlen argument value and the number of bytes sent by value.
2. Copies data sent by reference, if any, from the sender's message buffer to the caller's rbuf area. The number of bytes copied is the lesser of the rlen value and the number of bytes sent by reference.
3. Copies the priority of the packet and the number of bytes sent by value from the packet header to the pri and vxlen fields of the receiver's information-return area.
4. Copies the message reference, if any, contained in the packet to the corresponding three words of the receiver's information-return area. If the packet contains no message reference, the fourth word of the receiver's information-return area (rxlen) is zeroed.
5. Deallocates the packet, returning it to the kernel's free-element pool for reuse.

RCVCS then returns control to the caller, with the kernel-defined value TRUE in R0.

If no packet is queued on the specified semaphore at the time of the call, RCVCS returns immediately to the caller, with the kernel-defined value FALSE in R0, indicating that the Conditional Receive operation was unsuccessful.

#### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

The packet format expected by RCVCS is shown in Figure 3-1 (in Section 3.33).

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Error Returns

- F.IIID - Illegal semaphore identifier or name; not a valid queue semaphore.
- E.ILPM - Illegal parameter; rtnptr is not a word-address (even) value.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.28 RCVDS (RECEIVE DATA)

Pascal equivalent: RECEIVED Procedure  
RECEIVE\_ACE Procedure

The Receive Data (RCVDS) primitive, the complement of the SENDS primitive, permits any type of process to receive data sent through a queue packet. This primitive is the analog of WAQS for use by general and limited-system processes, which cannot access a packet directly in a mapped environment.

RCVDS performs the same wait operation on a specified queue semaphore as performed by WAQS but also copies the data contained in the packet to the caller's buffer space instead of returning the packet pointer. The packet format expected by RCVDS is the same as that produced by SENDS or SNDSCS, as shown in Figure 3-1. After the data is copied, the packet is returned to the kernel's free-element pool for reuse. (See the SGLQS primitive for further information about packets.)

A message sent by value (up to 34 bytes in length) is copied by the RCVDS primitive from the packet to the receiver's buffer. In the case of a message sent by reference, possibly longer than 34 bytes, the message is ordinarily copied from the sender's message buffer, which is described in the packet, to a buffer specified by the receiver. If no by-reference buffer is specified in the receive request, the message is not copied, but the primitive returns the message reference to the caller.

The message-by-reference feature must be used with caution concerning the length of individual messages. Since message copying is done within the Receive primitive, no other process can gain control until the entire message has been copied, because kernel primitive operations are indivisible. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

See RCVCS for the conditional (nonblocking) form of receive request.

#### Syntax

The three variants of the RCVDS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
RCVDS	RCVDS (area,sdb,rtnptr,vlen,vbuf,rlen,rbuf)
RCVDSS	RCVDSS (sdb,rtnptr,vlen,vbuf,rlen,rbuf)
RCVDSP	RCVDSP (sdb,rtnptr,vlen,vbuf,rlen,rbuf)

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

16RFA=base+blk-address

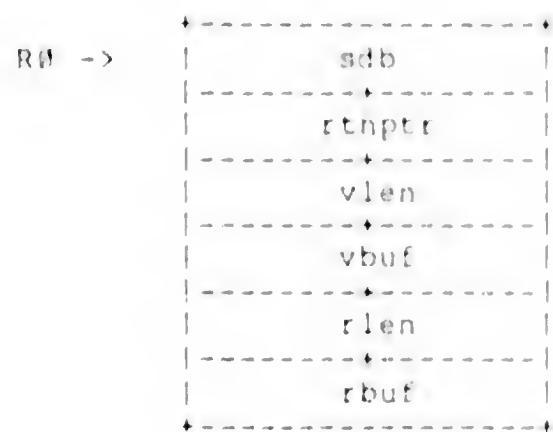
## MACRO-11 PRIMITIVE SERVICE REQUESTS

sdb	The address of the structure descriptor block (SDB) that identifies the queue semaphore to be waited on. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=] sdb-address
rtnptr	The address of a 4-word area in which information about the Receive operation is to be returned by the primitive. The format of the information returned is shown below. This argument has the form:  [RTNPTR=] word-address
vlen	The length, in bytes, of the buffer pointed to by vbuf. This argument limits the amount of by-value data (if any) to be copied from the packet. The value of this argument can range from 0 to 34. This argument has the form:  [VLEN=] integer
vbuf	The address of the buffer area in which data sent by value is to be copied. This argument has the form:  [VBUF=] address  This argument is significant only if vlen is nonzero.
rlen	The length, in bytes, of the buffer pointed to by rbuf. This argument limits the amount of by-reference data, if any, to be copied from the sender's buffer. If the value of this argument is 0 and if a message reference exists in the packet, the message is not copied; the reference is returned in the area pointed to by rtnptr, however. This argument has the form:  [RLEN=] integer
rbuf	The address of the buffer area in which data sent by reference is to be copied. This argument has the form:  [RBUF=] address  This argument is significant only if rlen is nonzero.

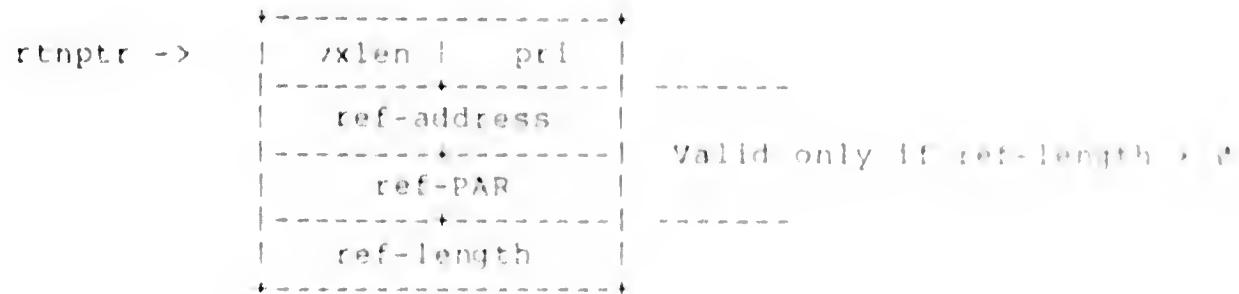
**Restrictions -** The value of parameter rlen may not exceed 8120.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the RCVDSx macros has the following format:



**Format of Information Returned** - The information returned to the caller in the 4-word area pointed to by rtnptr is in the following form:



<b>Parameter</b>	<b>Definition</b>
<b>pri</b>	The priority value that was assigned to the packet by the send operation.
<b>vxlen</b>	The number of bytes sent by value. (Note that this value may be greater than the number of bytes received, which is limited by the vlen argument.) If 0, no data was sent by value.
<b>ref-address</b>	The address of the sender's by-reference buffer, if any. This return value is valid only if the ref-length return value is nonzero; otherwise, the contents of this word are unpredictable.
<b>ref-PAR</b>	The value of the page address register that maps the sender's by-reference buffer, if any. This return value is valid only if the ref-length return value is nonzero; otherwise, the contents of this word are unpredictable.
<b>ref-length</b>	The number of bytes sent by reference. (Note that this value may be greater than the number of bytes received, which is limited by the rlen argument.) If 0, no data was sent by reference.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
RCVD$S sdb=#AQSEM,rtnptr=#RTNBLK,vlen=#0,vbuf=#0,rlen=#200,  
rbuf=#LONGB
```

### Semantics

The RCVD\$ primitive performs a wait operation on the specified queue semaphore, as described for the WAIQ\$ primitive, which may cause the calling process to block until a packet is available. When a packet is obtained from the semaphore's packet queue, the primitive performs the following actions, as governed by the arguments specified in the call:

1. Copies data sent by value, if any, from the packet in kernel space to the caller's vbuf area. The number of bytes copied is the lesser of the vlen argument value and the number of bytes sent by value.
2. Copies data sent by reference, if any, from the sender's message buffer to the caller's rbuf area. The number of bytes copied is the lesser of the rlen argument value and the number of bytes sent by reference.
3. Copies the priority of the packet and the number of bytes sent by value from the packet header to the pri and vxlen fields of the receiver's information-return area.
4. Copies the message reference, if any, contained in the packet to the corresponding three words of the receiver's information-return area. If the packet contains no message reference, the fourth word of the receiver's information return area (rxlen) is zeroed.
5. Deallocates the packet, returning it to the kernel's free-element pool for reuse.

RCVD\$ places the calling process in ready-active state if it was blocked by the wait operation or returns control to the caller if it was not blocked.

The packet format expected by RCVD\$ is shown in Figure 3-1 (in Section 3.33).

### Error Returns

- E.IIID - Illegal semaphore identifier or name; not a valid queue semaphore.
- E.ILPM - Illegal parameter; rtnptr is not a word-address (even) value.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.29 REXCS (REPORT EXCEPTION)

Pascal equivalent: REPORT Procedure

The Report Exception (REXCS) primitive is used to report a software exception condition or to simulate -- force -- a hardware exception. Any type of exception can be reported with this primitive.

See the SERAS primitive and Chapter 7 for a description of exception stack frames.

#### Syntax

The three variants of the REXCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

##### Variant

##### Macro Call

REXCS

REXCS [area,mask,subcod,arglen,argbuf]

REXCSS

REXCSS [mask,subcod,arglen,argbuf]

REXCSP

REXCSP [mask,subcod,arglen,argbuf]

##### Parameter

##### Definition

area

The address of a user-memory area in which the calling argument block is to be constructed (or found). This argument has the form:

[AREA=]arg-blk-address

mask

The type of exception, as indicated by predefined bit-mask symbols, to be reported. The exception-type symbols are described in Section 7.1. The symbols are defined by the EXMSKS macro. This argument has the form:

[MASK=]symbol

subcod

The subcode for the exception type, as indicated by predefined bit-mask symbols. The subcode symbols are described in Section 7.1. The symbols are defined by the EXMSKS macro. This argument has the form:

[SUBCOD=]symbol

If a given exception type has no applicable subcodes, the symbol ESSNSC (no subcode) can be used.

arglen

The address of a word containing the length, in bytes, of the user's exception argument buffer. This argument has the form:

[ARGLEN=]word-address

argbuf

The address of the user's exception argument buffer. This argument has the form:

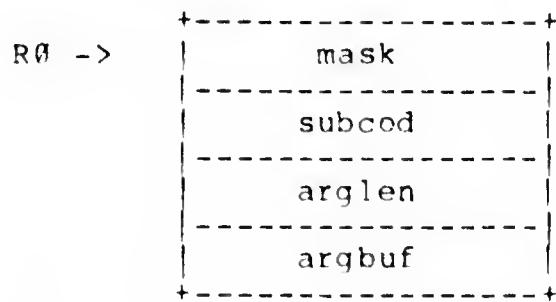
[ARGBUG=]buffer-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Restrictions** - The exception argument buffer must begin on a word boundary. The argument buffer length must be an even number of bytes.

**Exception-Types Symbols** - See Section 7.1. The symbols are defined by the EXMSKS macro.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the REXCSx macro has the following format:



### Syntax Example

```
REXCS area=#ARGBLK,subcod=#EX$US1,arglen=#SUMNUM,argbuf=#VALUES
```

### Semantics

The REXCS primitive causes the kernel's exception dispatcher to be entered for normal disposition of the exception condition indicated by mask. The exception condition is dispatched to an exception handler, if one exists for the exception type and the reporting process's exception group, or is dispatched to the reporting process if possible. In either case, an exception stack frame is placed on the reporting process's stack; the frame contains the contents and length of the argument buffer specified in the service request, as well as other standard information (see SERAS and Chapter 7).

### Error Returns

E.ILPM - Illegal parameter; more than one exception type was specified, no exception type was specified, or the argument buffer length value is odd.

E.ADDR - Illegal address; the argument buffer address either was not on a word boundary or was not in the process's address space.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.30 RSUM\$ (RESUME PROCESS)

Pascal equivalent: RESUME Procedure

The Resume Process (RSUM\$) primitive reactivates a suspended process, assuming that the process's suspension count was -1, which implies current suspension but no further suspensions pending. (The primitive increments the process's suspension count and returns a TRUE or FALSE indication, as described in the Semantics section below.) This allows the calling process to unsuspend another process (see the SPNDS primitive).

#### Syntax

The three variants of the RSUM\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
---------	------------

RSUM\$	RSUM\$ [area,pdb]
--------	-------------------

RSUM\$S	RSUM\$S [pdb]
---------	---------------

RSUM\$P	RSUM\$P [pdb]
---------	---------------

Parameter	Definition
-----------	------------

area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:
------	--

[AREA=]arg-blk-address

pdb	The address of the process descriptor block (PDB) that identifies the process to be resumed. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:
-----	---

[PDB=]pdb-address

**Argument Block** - The calling argument block generated - or assumed to exist -- by the RSUM\$X macro has the following format:



#### Syntax Example

RSUM\$S pdb=#APROC

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Semantics

The RSUMS primitive increments the suspension count associated with the specified process. If the count changes from -1 to 0, the state of the process is changed to ready active, wait active, or exception-wait-active, depending on its state at the time of resumption. The scheduler is called if the new state is in fact ready active; otherwise, the primitive returns to the caller after incrementing the count.

Because of the suspension counter, a Resume request may not actually reactivate a process. (For example, if the suspension count was positive, the process was already active, and no state transition occurs.) The immediate effect of the request is indicated as a TRUE or FALSE function return (in R0). A TRUE return indicates that the process either was reactivated -- changed from the suspended state to the appropriate active state -- or was already active. A FALSE return indicates that the process was not changed from the suspended state, because the suspension count remained negative after the increment.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

The maximum value of the suspension counter is 32,767; the minimum value is -32,768. Thus, the counter can record a maximum of 32,768 successive Suspend requests or 32,767 successive Resume requests.

### Error Returns

E.IID - Invalid process description (index or name); no such process exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.31 SALLS (SIGNAL ALL WAITERS)

Pascal equivalent: SIGNAL\_ALL Procedure

The Signal All Waiters (SALLS) primitive unblocks all processes waiting on a specified binary or counting semaphore, leaving that semaphore at 0. If no process is waiting on the semaphore, the SALL operation leaves the value of the semaphore variable unchanged.

This permits the calling process to signal simultaneously all processes that are waiting for the same event to occur and to ensure that the semaphore is left in the closed state in all cases. (Compare with SGLCS, the conditional signal.)

Together, the WAITS and SALLS calls allow two or more processes to synchronize on a single event signaled by another process.

#### Syntax

The three variants of the SALLS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.1.)

Variant	Macro Call
SALLS	SALLS [area,sdb]
SALLSS	SALLSS [sdb]
SALLSP	SALLSP [sdb]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of a structure descriptor block (SDR) that identifies the semaphore to be signaled. (See Section 3.1.5 for the format and use of an SDR.) This argument has the form:  [SDR=]sdb-address

**Restrictions** - The semaphore identified by the passed SDR must not be a queue semaphore.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SALLSx macro has the following format:

```
RR -> +-----+
          |       sdb      |
          +-----+
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
SALL$ sdb=ASYNC
```

### Semantics

The SALL\$ primitive unblocks all processes waiting on the specified semaphore, sets the semaphore variable to 0, and calls the scheduler. If no process is waiting on the specified semaphore, the SALL\$ primitive leaves the value of the semaphore variable unchanged and returns control to the caller.

The SALL\$ request may cause the calling process to be preempted, depending on the priorities of the processes unblocked by the operation.

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such binary or counting semaphore exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.32 SCHD\$ (SCHEDULE PROCESS)

Pascal equivalent: SCHEDULE Procedure

The Schedule Process (SCHD\$) primitive switches the calling process out of the run state if another process of equal priority is eligible for control of the CPU.

This primitive permits a process to choose when to relinquish the CPU to some other equal-priority process, thus circumventing the kernel's event-triggered scheduling mechanism. (See the Applications section below.)

#### Syntax

The SCHD\$ macro call syntax is:

```
SCHD$
```

#### Semantics

If the first process in the ready-active queue has the same priority as the caller, the SCHD\$ primitive places the calling process on the ready-active queue, behind all processes of the same priority, and calls the scheduler. This causes the first process in the queue to be placed in the run state.

If there is no ready-active process of equal priority at the time of the call, the primitive returns immediately to the caller.

#### Error Returns

None.

#### Applications

The SCHD\$ primitive may be used in systems with several processes of the same priority, as when several instances of a process are used to create the effect of concurrency. If the task performed by a process tends to be compute-bound, that process uses an inequitable share of the processor resource. This can be avoided by inserting a SCHD\$ request at an appropriate location in the process. The resulting preemption will place the current process behind other processes of the same priority in the ready-active queue, giving them a chance to execute.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.33 SEND\$ (SEND DATA)

Pascal equivalent: SEND Procedure  
Pascal variant: SEND\_ACK Procedure

The Send Data (SEND\$) primitive allocates a queue packet in kernel space, copies user data into the packet, and signals a specified queue semaphore. The signal operation is the same as that performed by SGLQ\$. This primitive is the analog of SGLQ\$ for use by general and device-access processes, which cannot access a packet directly in a mapped environment. SEND\$ permits any type of process to transmit data to another process through a packet. The complementary Receive Data (RCVDS) primitive permits any type of process to receive data sent through a packet. (See the SGLQ\$ primitive for further information about packets.)

A limited amount of data -- up to 34 bytes -- can be sent by value; that is, a short message can be sent directly in the packet. A larger amount of data can be sent by reference, or indirectly; a reference to the message, not the message itself, is sent in the packet. These two methods can also be combined in one Send request; some data can be sent by value and some by reference.

The by-reference feature permits messages that are too large to fit into a packet to be exchanged between two processes with one Send and one Receive request. The address and length of the message buffer and the buffer PAR value are placed in the packet for subsequent use by the RCVDS primitive. The message is copied -- from the sender's buffer to the receiver's buffer -- only when the corresponding receive request is issued.

As an alternative to having a message by reference copied by the RCVDS primitive, the receiver can specify that only the message reference is to be returned. The message-by-reference feature must be used with caution concerning the length of individual messages. Since message copying is done within the Receive primitive, no other process can gain control until the entire message has been copied, because kernel primitive operations are indivisible. Thus, transmission of long messages can seriously affect the servicing of interrupts by increasing interrupt latency throughout the system.

See SNDC\$ for the conditional signal form of the Send request.

#### Syntax

The three variants of the SEND\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SEND\$	SEND\$ [area,sdb,pri,vlen,vbuf,rbuf,rbuf]
SEND\$S	SEND\$S [sdb,pri,vlen,vbuf,rbuf,rbuf]
SEND\$P	SEND\$P [sdb,pri,vlen,vbuf,rbuf,rbuf]

## MACRO-11 PRIMITIVE SERVICE REQUESTS

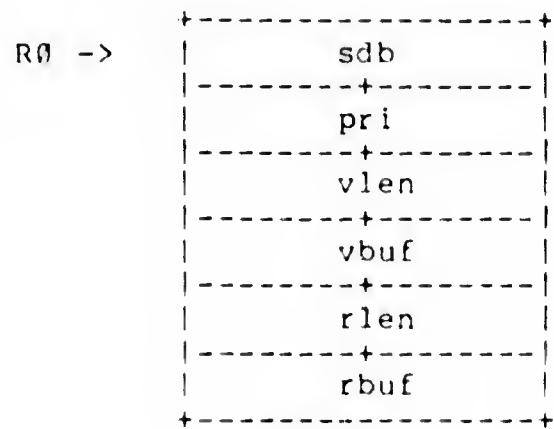
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of the structure descriptor block (SDB) that identifies the queue semaphore to be signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=]sdb-address
pri	The priority value (0 to 255) to be assigned to the packet, which affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see CRSTS\$). This argument has the form:  [PRI=]integer
vlen	The number of bytes to be transmitted by value. The maximum is 34 if no message is sent by reference (if rlen = 0) and 28 if a message reference is specified. This argument has the form:  [VLEN=]integer
vbuf	The address of a message buffer to be transmitted by value. The content of this buffer is copied into the packet directly. This argument has the form:  [VBUF=]buffer-address  This argument is significant only if vlen is nonzero.
rlen	The length of a message to be sent by reference -- the length of the message buffer pointed to by rbuf. If nonzero, this value is placed in the packet along with the rbuf value, following any data sent by value. This argument has the form:  [RLEN=]integer
rbuf	The address of a buffer containing a message to be sent by reference. This address is placed in the packet as a 2-word physical address by the addition of a user PAR value. (See Figure 3-1.) This argument has the form:  [RBUF=]buffer-address  This argument is significant only if rlen is nonzero.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Restrictions** - Thirty-four bytes are available in a queue packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

The value of parameter rlen may not exceed 8129.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SEND\$X macros has the following format:



### Syntax Example

```
SEND$ area=#ARGBLK,sdb=#MSGSEM,pri=#0,vlen=#34.,vbuf=#MSG,  
rlen=#100.,rbuf=#MSG+34.
```

### Semantics

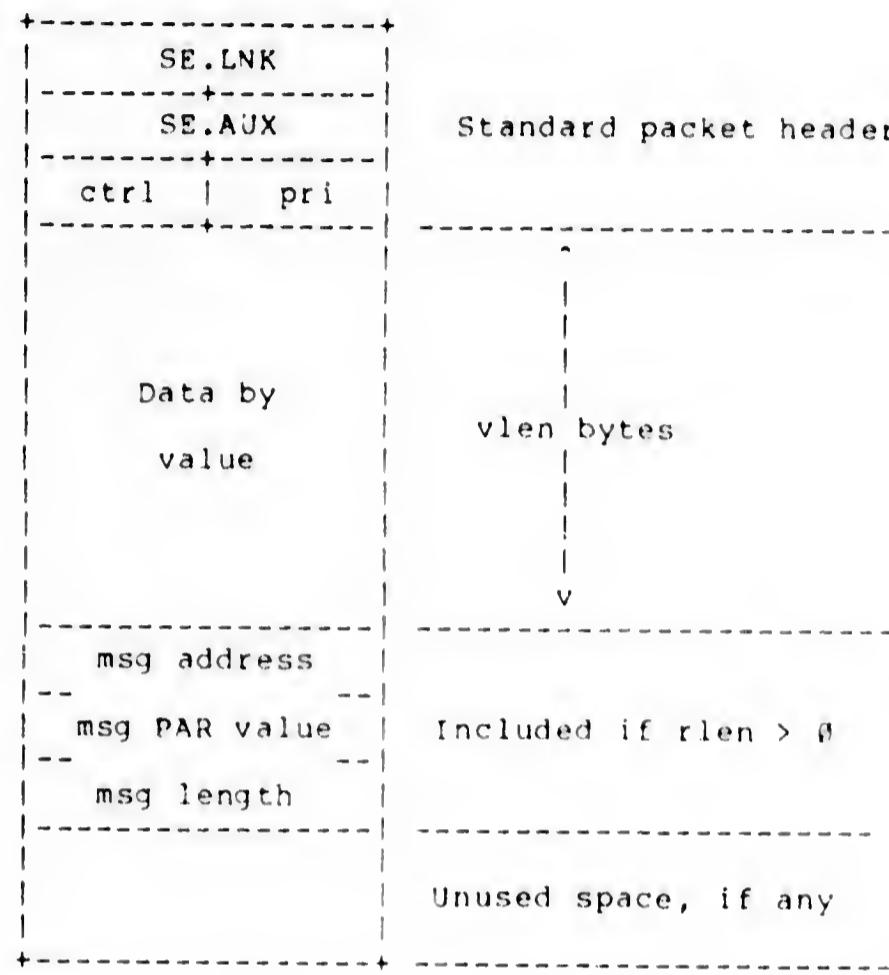
The SEND\$ primitive performs the following actions before signaling the specified queue semaphore:

1. Obtains a packet -- system element -- from the kernel's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte based on the value-size and reference-length arguments and places it in the packet header for subsequent use by RCVDS\$.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in kernel space.
4. Constructs a physical address from the address (rbuf) of the message to be sent by reference, if any. This physical address is placed in the packet along with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode PAR associated with that virtual address.

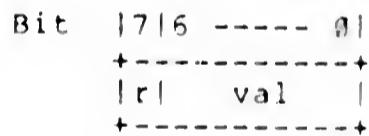
SEND\$ then signals the semaphore, placing the packet on the semaphore's queue, as described for the SGLQ\$ primitive.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

The format of a packet constructed by SEND\$ (or by SNDC\$) is shown in Figure 3-1.



The format of the byte indicated by ctrl is:



where

r = 0 if no message reference  
= 1 if reference is included

val = number of bytes by value  
(7-bit integer)

Figure 3-1 SEND\$/SNDC\$ Packet Format

Note that there are synchronization differences between data sent by value and data sent by reference. That is, data sent by value is copied immediately, thus freeing the buffer it occupied for immediate use. Data sent by reference is not copied until the corresponding RCVx initiative is executed, and thus the buffer it occupies is not free for use until then.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Error Returns

E.IIID - Illegal semaphore identifier or name; not a valid queue semaphore.

E.ILPM - Invalid parameter; amount of data to be sent by value (vlen parameter) exceeds packet capacity

### Applications

SEND\$ is the basic buffer-transfer mechanism supplied by the kernel. SEND\$ provides a primitive message exchange mechanism for use between general processes or between general and privileged processes. For example, it is used to implement the interface to higher-level services such as those provided by the clock and device-handler processes. This interface consists of a request message sent to the appropriate system process and a reply received from the process, using the SEND\$/RCV\$ facility.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.34 SERA\$ (SET EXCEPTION ROUTINE ADDRESS)

Pascal equivalent: ESTABLISH Procedure  
REVERT Procedure

The Set Exception Routine Address (SERA\$) primitive establishes an exception entry point within the calling process for a specified set of exception conditions. The exception address and set of conditions thus established replace any that were inherited from a parent process.

This primitive allows a process to regain control after causing a particular type of exception, in either of two circumstances:

1. No exception handler exists for the type of exception and exception group of the process involved.
2. The existing exception handler chooses to pass the exception on to the faulting process (via the DEXC\$ primitive).

When the process causing the exception is reentered at its exception address, the user's stack contains information describing the condition.

#### Syntax

The three variants of the SERA\$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SERA\$	SERA\$ [area,adr,mask]
SERA\$S	SERA\$S [adr,mask]
SERA\$P	SERA\$P [adr,mask]

#### Parameter      Definition

area      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

'AREA=larg-blk-address

adr      The address of the exception routine for the process, or 0. (See the Argument Block section below.)

## MACRO-11 PRIMITIVE SERVICE REQUESTS

mask        The type(s) of exception, as indicated by predefined bit-mask symbols, to be received by this process. The exception-type symbols, of the form EX\$xxx, are described in Section 7.1.

The type symbols may be ORed as desired. These symbols are defined by the EXMSK\$ macro. This argument has the form:

[MASK=]symbol[!symbol]

### NOTE

If the adr argument value is 0, the meaning of the request changes to "disable exception address" for the calling process. That is, a call specifying an exception address of 0 cancels any previous SERA\$ call made by the process and disables the passing of any exceptions to the process. The mask argument is not meaningful in this case.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SERA\$x macro has the following format:

R0 ->	+	-----+              adr               -----+-----               mask              +-----+-----+
-------	---	---

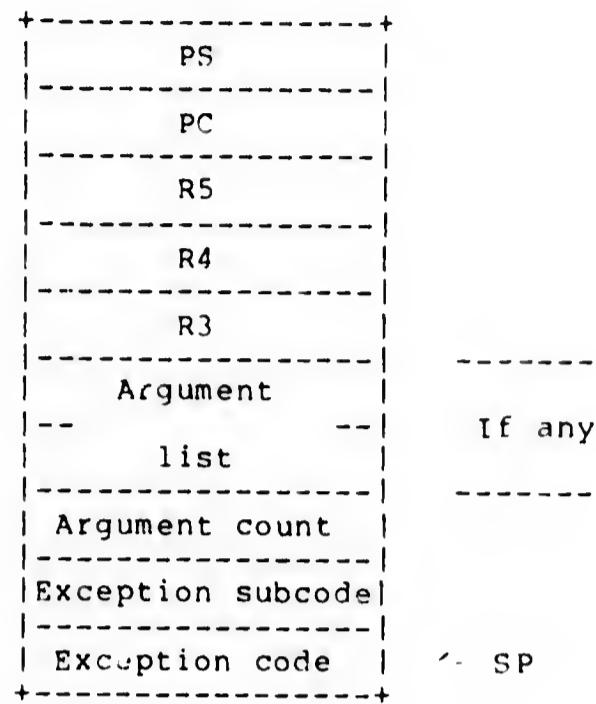
### Semantics

The SERA\$ primitive places the specified exception address and exception bit mask in the caller's PCB, replacing the exception routine, if any, inherited when the process was created. The primitive then returns to the caller.

When the kernel passes an exception back to the process that caused it (see the CCND\$ and DEXC\$ semantics), the exception service routine is entered in full process context, as though a software interrupt of the process had occurred. The user stack contains the PS and PC of the exception, the content of registers 3, 4, and 5 at the point of exception, a list of arguments -- dependent on the exception type -- an argument count, and the exception code. (The exception code is a bit mask corresponding to the exception type, as for the mask argument.)

## MACRO-11 PRIMITIVE SERVICE REQUESTS

On entry to the exception routine, the stack looks as follows:



The exception routine must save and restore R0, R1, and R2 if it uses them, restore R3, R4, and R5 from the stack, purge the stack down to the RC value, and exit with an RTI instruction.

The arguments, if any, and the argument count, in bytes, passed on the stack for each type of exception are described in Chapter 7.

### Error Returns

E.ADDR - Illegal exception routine address; an odd value.

E.ILPR - Illegal primitive call; SERA called from FORK level.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.35 SGLCS (CONDITIONALLY SIGNAL SEMAPHORE)

Pascal equivalent: COND\_SIGNAL Function

The Conditionally Signal Semaphore (SGLCS) primitive signals a specified binary or counting semaphore if at least one process is already waiting on that semaphore. If no process is waiting, the semaphore is not signaled -- variable value is not incremented -- and the kernel-defined value FALSE is returned to the caller in R0.

If the semaphore is signaled, the first process waiting on the semaphore is unblocked, and the kernel-defined value TRUE is returned to the caller in R0. This permits the calling process to signal another process that an event it is waiting on has occurred, but only if the wait request is issued before the signal. This in turn allows the caller to selectively signal of a set of semaphores, unblocking one process, if any, waiting on a semaphore of that set.

Note that unlike the SGNLS primitive, the order in which the signal and wait occur affects the operation of the SGLCS primitive. The SGLCS call provides, for example, a means of testing each of a set of identical server processes for availability. (See also WAICS.)

#### Syntax

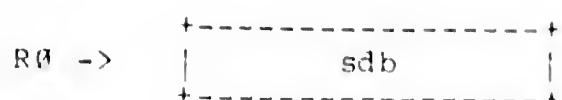
The three variants of the SGLCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SGLCS	SGLCS [area,sdb]
SGLCSS	SGLCSS [sdb]
SGLCSP	SGLCSP [sdb]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of a structure descriptor block (SDB) that identifies the semaphore to be conditionally signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=]sdb-address

**Restrictions** - The semaphore identified by the passed SDB must not be a queue semaphore.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SGLC\$X macro has the following format:



### Syntax Example

SGLCSS R2

### Semantics

The SGLCS primitive signals the specified semaphore only if at least one process is waiting. Otherwise, the primitive returns immediately to the caller, with the kernel-defined value FALSE in R0, indicating that the semaphore was not signaled.

If the signal operation succeeds, the primitive switches the first waiting process to the ready-active state, decrements the semaphore value, and calls the scheduler. This may cause the calling process to be switched to the ready-active state -- lose control of the CPU -- depending on the relative priority of the process at the head of the ready-active queue. On eventual return to the caller, R0 contains the kernel-defined value TRUE, indicating that the semaphore was signaled.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

CMP #TRUE,R0

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such binary or counting semaphore exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.36 SGLQS (SIGNAL QUEUE SEMAPHORE)

Pascal equivalent: PUT\_PACKET Procedure

The Signal Queue Semaphore (SGLQS) primitive places a packet on a specified semaphore's packet queue and signals the semaphore. If any processes are waiting on that semaphore, the first waiting process is unblocked, and a pointer to the packet is eventually passed to that process. (The packet is dequeued in this case.) If no process is waiting, the packet remains on the queue, and the signal remains in effect.

This primitive permits the calling process to signal another process that a data packet it needs, or will need, is available, regardless of whether the other process is already waiting for the signal. (Compare with SGQS\$, the Conditional Signal Queue call.)

A packet is a fixed-length system data structure that is allocated by the kernel from a special system-memory pool (see the ALPK\$ primitive and Section 2.2.2). The overall size of a packet, including the 3-word header, is given by the global symbol SE.SIZ, in bytes. The length of the undefined, arbitrarily usable portion of the packet is given by the global symbol QE.LEN, also in bytes. The value of QE.LEN is 34(decimal) for the MicroPower/Pascal system as distributed. Privileged processes can obtain an empty packet by means of an Allocate Packet (ALPK\$ or ALPcs) request. (The packet is not initialized to zeros.) A packet can be returned to the kernel's free-packet pool through use of the DAPK\$ primitive.

General and device-access processes do not have direct access to a packet -- for example, may not store into one directly -- in a mapped environment. Therefore, if such a process needs to send data via a packet, as opposed to passing one along that it has acquired via another primitive, it must use the SEND\$ primitive. This primitive provides a packet-acquisition and data-copying service in addition to the functionality of SGLQS\$.

The WAIQS call is the inverse of the SGLQS call.

#### Syntax

The three variants of the SGLQS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SGLQS	SGLQS [area,sdb,qelm]
SGLQSS	SGLQSS [sdb,qelm]
SGLQSP	SGLQSP [sdb,qelm]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**sdb** The address of a structure descriptor block (SDB) that identifies the semaphore to be signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=] sdb-address

**qelm** The address of a pointer to the packet that is to be placed on the semaphore queue. This argument has the form:

[QELM=] pointer-address

**Restrictions** - The semaphore identified by the passed SDB must be a queue semaphore.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SGLQSx macro has the following format:



### Syntax Example

SGLQS\$  sdb=#QSEM,qelm=pktptr

### Semantics

The SGLQS primitive tests the specified queue semaphore for waiting processes. If no process is waiting, the primitive signals the semaphore, links the passed packet into the packet queue, and returns to the caller.

If at least one process is waiting, the primitive unblocks the first waiting process, associates the passed packet pointer with that process -- as its wait-return value -- and calls the scheduler. This may cause the calling process to be preempted, depending on the priority of the unblocked process.

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such queue semaphore exists.

### Applications

Queue semaphores may be used to implement general queueing functions. The SGLQS operation places a packet on a queue, where it remains until another process removes it with either a wait or a receive operation. This basic mechanism can be used to implement a simple message facility or a generalized queued I/O facility.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.37 SGNLS (SIGNAL SEMAPHORE)

Pascal equivalent: SIGNAL Procedure

The Signal Semaphore (SGNLS) primitive signals a specified binary or counting semaphore, unblocking the first process, if any, waiting on that semaphore. This permits the calling process to signal to another process that an event has occurred, regardless of whether the other process is already waiting for the signal. (Compare with SGLCS, the conditional signal.)

Or, viewing a binary semaphore as a gate, the signal primitive permits the calling process to open the gate for another process that it has previously closed behind itself with a WAITS call. In the latter case, the WAITS might be issued before, and the SGNLS issued after, entering a critical section of code -- critical relative to the operation of a closely related process.

Together, the WAITS and SGNLS calls allow two or more cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms. (See also SGLCS, WAITS, and SALLS.)

#### Syntax

The three variants of the SGNLS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SGNLS	SGNLS [area,sdbl]
SGNLSS	SGNLSS [sdbl]
SGNLSP	SGNLSP [sdbl]

area                      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

sdb                      The address of a structure descriptor block (SDB) that identifies the semaphore to be signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=] sdb-address

**Restrictions** - The semaphore identified by the passed SDR must not be a queue semaphore.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SGNL\$X macro has the following format:

```
R0 -> +-----+  
          |      sdb     |  
          +-----+
```

### Syntax Example

```
SGNL$ area=#ARGBLK,sdb=#CSEM
```

### Semantics

The SGNL\$ primitive signals a binary semaphore -- increments the semaphore's gate variable -- if it is 0 or returns immediately to the caller if it is 1. The SGNL\$ primitive signals a counting semaphore -- increments the semaphore's counter variable -- and, if its previous value was greater than 0, returns immediately to the caller.

In either case, if the signal causes the semaphore value to change from 0 to 1 and if at least one process is waiting on the semaphore, the primitive unblocks the first waiting process, decrements the semaphore value, and calls the scheduler. This may cause the calling process to be preempted, depending on the priority of the unblocked process. If the semaphore value changes from 0 to 1 and if no process is waiting, the primitive returns immediately to the caller.

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such binary or counting semaphore exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.38 SGQCS (CONDITIONALLY SIGNAL QUEUE SEMAPHORE)

Pascal equivalent: COND\_PUT\_PACKET Function

The Conditionally Signal Queue Semaphore (SGQCS) primitive signals a specified semaphore only if at least one process is waiting on that semaphore. If so, the first waiting process is unblocked, and a pointer to the packet passed by the caller is returned to that process. The kernel-defined value TRUE is eventually returned to the caller in R0. If no process is waiting, the primitive returns immediately to the caller with the kernel-defined value FALSE in R0.

This primitive permits the calling process to pass a data packet to another process, but only if the other process is already waiting for the packet. (Compare with SGLQS, the unconditional signal queue call.)

See the SGLQS primitive for a description of queue packets. General and device-access processes do not have direct access to a packet -- may not store into one directly -- in a mapped environment. Therefore, if such a process needs to send data via a packet as opposed to passing one along that it has acquired via another primitive, it must use the SNDQS primitive, which provides a packet-acquisition and data-copying service in addition to the functionality of SGQCS.

The WAQCS call is the inverse of the SGQCS call.

#### Syntax

The three variants of the SGQCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SGQCS	SGQCS [area,sdb,qelm]
SGQCSS	SGQCSS [sdb,qelm]
SGQCSP	SGQCSP [sdb,qelm]

#### Parameter      Definition

**area**      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

**sdb**      The address of a structure descriptor block (SDB) that identifies the semaphore to be conditionally signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=]sdb-address

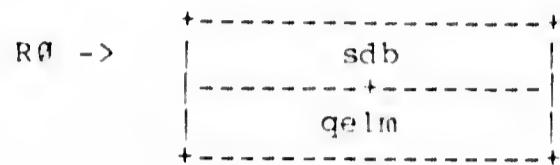
**qelm**      The address of a pointer to the packet that is to be passed via the queue semaphore. This argument has the form:

[QELM=]pointer-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Restrictions** - The semaphore identified by the passed SDB must be a queue semaphore.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SGQC\$X macro has the following format:



### Syntax Example

```
SGQC$S  sdb=#QSEM,qelm=R5
```

### Semantics

The SGQC\$ primitive tests the specified semaphore for waiting processes.

If at least one process is waiting on the semaphore, the primitive switches the first waiting process to the ready-active state, associates the passed packet pointer with that process as its wait-return value, and calls the scheduler. This may cause the calling process to be switched to the ready-active state -- lose control of the CPU -- depending on the relative priority of the process at the head of the ready-active queue. On eventual return to the caller, R0 contains the kernel-defined value TRUE, indicating a successful operation.

If no process is waiting on the semaphore, the primitive returns immediately to the caller, with the kernel-defined value FALSE in R0, indicating an unsuccessful operation.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such queue semaphore exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Applications

This primitive permits a process to selectively send a record -- queue packet -- to any of several queue semaphores if another process is already waiting for the record. For example, suppose that a process wishes to send an output request, contained in a packet, to any of three output-service processes associated with separate queue semaphores. The SGQCS call allows the submitting process to test each semaphore for an output process that is ready to service the request.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.39 SNDCS (CONDITIONAL SEND DATA)

Pascal equivalent: COND\_SEND Function  
COND\_SEND\_ACK Function

The Conditional Send Data (SNDCS) primitive implements a selective form of the SENDS operation. SNDCS allocates a queue packet, copies user data into it, and signals a specified queue semaphore, but only if at least one process is waiting on the semaphore. If the send operation was performed, the primitive returns the kernel-defined value TRUE in R0. If it was not, the kernel-defined value FALSE is returned in R0. The packet-allocation and data-copying portion of the operation is not done if there is no waiting process.

This primitive is the analog of SGQCS for use by general and device-access processes, which cannot access a packet directly in a mapped environment. SNDCS permits any type of process to conditionally transmit data to another process through a packet. The complementary RCVDS and RCVCS primitives permit any type of process to receive the data sent through a packet. (See the SGLQS primitive for further information about packets.)

The message-transmission features of SNDCS are identical to those provided by SENDS -- the sending of messages by value or by reference. The only functional difference between the two primitives is the unconditional signal performed by SENDS versus the conditional signal performed by SNDCS.

#### Syntax

The three variants of the SNDCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SNDCS	SNDCS [area,sdb,pri,vlen,vbuf,rlen,rbuf]
SNDCSS	SNDCSS [sdb,pri,vlen,vbuf,rlen,rbuf]
SNDCSP	SNDCSP [sdb,pri,vlen,vbuf,rlen,rbuf]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of the structure descriptor block (SDB) that identifies the queue semaphore to be signaled. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=]sdb-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**pri**

The priority value (0 to 255) to be assigned to the packet, which affects the order in which the packet is queued on a semaphore having a priority-ordered packet queue (see CRSTS). This argument has the form:

[PRI=]integer

**vlen**

The number of bytes to be transmitted by value. The maximum is 34 if no message is sent by reference -- that is, if rlen = 0 -- or 28 if a message reference is specified. This argument has the form:

[VLEN=]integer

**vbuf**

The address of a message buffer to be transmitted by value. The content of this buffer is copied into the packet directly. This argument has the form:

[VBUF=]buffer-address

This argument is significant only if vlen is nonzero.

**rlen**

The length of a message to be sent by reference, that is, the length of the message buffer pointed to by rbuf. If nonzero, this value is placed in the packet along with the rbuf value, following any data sent by value. This argument has the form:

[RLEN=]integer

**rbuf**

The address of a buffer containing a message to be sent by reference. This address is placed in the packet, together with rlen, as a 2-word physical address by addition of a user PAR value. This argument has the form:

[RBUF=]buffer-address

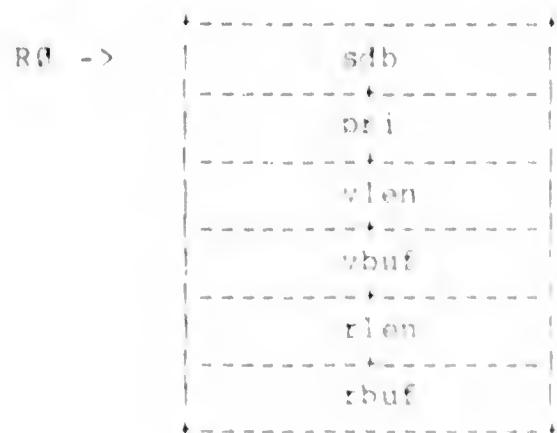
This argument is significant only if rlen is nonzero.

**Restrictions** - Thirty-four bytes are available in a queue packet for message data, a message reference, or both. A reference occupies three words in the packet and, if included, reduces the space available for data by value to 28 bytes.

The value of parameter rlen may not exceed 8129.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SEND\$X macros has the following format:



### Syntax Example

```
SNDCS area=#ARGBLK,sdb=#SERVOR,pri=#1A,vlen=$0,vbuf=#H,rlen=$200,
        rbuf=#LONG
```

### Semantics

The SNDCS primitive tests the specified queue semaphore for a waiting process and performs the following actions before signaling the semaphore:

1. Obtains a packet -- system element -- from the kernel's free-element pool and writes the specified priority value into the packet header.
2. Constructs a control byte based on the value-size and reference-length arguments and places it in the packet header for subsequent use by RCVNS.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in kernel space.
4. Constructs a physical address from the address (rbuf) of the message to be sent by reference, if any. This physical address is placed in the packet along with the message length. A physical address consists of two words. The value of the first word is the virtual address; the value of the second word is the content of the user-mode PAR associated with that virtual address.

SNDCS then signals the semaphore, unblocking the first waiting process and passing the packet to it. This may cause the calling process to be preempted, as described for the SGLQS primitive. On eventual return to the caller, RQ contains the kernel-defined value TRUE, indicating a successful operation.

If no process is waiting on the semaphore, the primitive returns immediately to the caller, with the kernel-defined value FALSE in RQ, indicating that the send operation was not performed.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

A packet constructed by SND\$CS has the same format as one constructed by the SEND\$ primitive, as shown in Figure 3-1.

### NOTE

It is possible for a process to be blocked waiting for a queue element in SND\$CS. This is different from other conditional forms of primitives that cannot block.

### Error Returns

- E.IIID - Illegal semaphore identifier or name; not a valid queue semaphore.
- E.ILPM - Invalid parameter; amount of data to be sent by value (vlen parameter) exceeds packet capacity.

### Applications

The SND\$CS primitive permits the sending process to be selective about message transmission. For example, if there are several equivalent service queues, a service-request message can be sent to the queue having an idle server process waiting for a request.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.40 SPNDS (SUSPEND PROCESS)

Pascal equivalent: SUSPEND Procedure

The Suspend Process (SPNDS) primitive places an active process in the suspended state if the process's suspension count was 0, which implies that no prior suspensions or resumptions were pending. (The primitive decrements the process's suspension count and returns a TRUE or FALSE indication, as described in the Semantics section below.)

This primitive allows the calling process to suspend either itself or another process. The suspended process is prevented from executing until resumed by some other process (see the RSUMS primitive). Together, the SPNDS and RSUMS primitives provide a mutual (or unilateral) exclusion mechanism more radical than that provided by the WAITS and SGNLS primitives.

#### Syntax

The three variants of the SPNDS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
SPNDS	SPNDS [area,pdb]
SPNDSS	SPNDSS [pdb]
SPNDSP	SPNDSP [pdb]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
pdb	The address of the process descriptor block (PDB) that identifies the process to be suspended. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:  [PDB=]pdb-address

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the SPNDSx macro has the following format:



## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
SPNDS$, sdb=#0 ; Suspend self
```

### Semantics

The SPNDS primitive decrements the suspension count associated with the specified process. If the count changes from 0 to -1, the state of the process is changed to either ready suspended or wait suspended, depending on its state at the time of suspension, and the scheduler is invoked if the suspended process was the caller. Otherwise, the primitive returns to the caller after decrementing the count.

The suspension counter may prevent a suspend request from suspending a process. The immediate effect of the request is indicated as a TRUE or FALSE function return (in R0). A TRUE return indicates that the process was changed to or was already in the suspended state. A FALSE return indicates that the process was not suspended by the suspend operation.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

Note that a suspend operation on a stopped process is always ineffective; see the STPCS primitive.

A transition from the wait-suspended state to the ready-suspended state can occur while a process is suspended. (A SGNLS operation can unblock a waiting suspended process, for example.) A resume operation is required to reactivate a suspended process, however.

The maximum value of the suspension counter is 32,767; the minimum value is -32,768. Thus, the counter can record a maximum of 32,768 successive Suspend requests or 32,767 successive Resume requests.

### Error Returns

E.IIID - Invalid process description (index or name); no such process exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.41 STPCS (STOP PROCESS)

Pascal equivalent: STOP Procedure

The Stop Process (STPCS) primitive stops a specified process by forcing its termination entry point when it is next reentered. If the subject process is either blocked or suspended at the time of the call, it is forced into the ready-active state, as described in the Semantics section below. The subject process has a special aborted status, which means that it cannot subsequently be suspended.

The STPCS primitive may return control to the calling process, depending on the relative priorities of the caller and the process to be stopped. (The calling process and the subject process may be one and the same.)

When the stopped process is reentered at its termination point, it must determine what it should do before stopping (deleting itself). Minimally, it should deallocate any owned resources: delete any semaphores or other structures that it created, return any packets to the kernel's free-packet pool, and so forth. Before resource deallocation, it could take any actions needed for a graceful termination, such as completing an in-progress I/O operation or message transmission. At the appropriate point, the process deletes itself from the system by issuing a DLPCS request.

#### Syntax

The three variants of the STPCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
STPCS	STPCS [area,pdb]
STPCSS	STPCSS [pdb]
STPC\$P	STPC\$P [pdb]

#### Parameter      Definition

area      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

pdb      The address of the process descriptor block (PDB) that identifies the process to be stopped, or 0. If 0 is specified, the calling process is implied. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:

[PDB=]pdb-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the STPC\$*x* macros has the following format:

```
R0 -> +-----+  
          |     pdb    |  
          +-----+
```

### Syntax Example

```
STPC$ area=#STPARG,pdb=#NGPROC
```

### Semantics

The STPC\$ primitive modifies the specified process's context so that it will begin execution at its termination entry point when it is subsequently scheduled for execution. The primitive also sets a special aborted status indication, which inhibits any later suspension of the subject process. If the process is in ready-active state, the primitive returns control to the calling process. If the subject process is not in ready-active state, one of the following cases applies:

1. If the subject process is blocked on a binary, counting, or queue semaphore -- waiting active state -- it is removed from the semaphore's waiting process list and placed on the ready-active queue. The primitive then calls the scheduler.
2. If the subject process is blocked on a ring buffer -- waiting active state -- it is removed from the ring buffer's waiting process list and placed on the ready-active queue. No adjustment is made for any partial transfer to or from the ring buffer that may have occurred on behalf of the subject process; no buffer resetting is done. The primitive then calls the scheduler.
3. If the subject process is in exception wait state, the primitive returns control to the caller. The subject process will be placed on the ready-active queue when the exception handler finishes processing the exception.
4. If the subject process is in any of the suspended states -- ready suspended, wait suspended, or exception wait suspended -- it is made active and then treated as described above.

### Error Returns

E.IID - Invalid process description (index or name); no such process exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.42 WAICS (CONDITIONALLY WAIT ON SEMAPHORE)

Pascal equivalent: COND\_WAIT Function

The Conditional Wait on Semaphore (WAICS) primitive decrements the value of a specified binary or counting semaphore if its current value is nonzero, indicating a previous signal, and returns to the caller with the kernel-defined value TRUE in R0. If the semaphore has not been signaled -- current value is already 0 -- the primitive returns immediately to the caller with the kernel-defined value FALSE in R0. In neither case is the calling process blocked, that is, made to wait until the semaphore is signaled. (Compare with WAITS, the unconditional form.)

This primitive permits the calling process to test for the arrival of a signal from another process without blocking when no signal has occurred. That is, the calling process proceeds in any case.

#### Syntax

The three variants of the WAICS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.1)

Variant	Macro Call
WAICS	WAICS [area,sdb]
WAICSS	WAICSS [sdb]
WAICSP	WAICSP [sdb]

Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]addr-blk-address
sdb	The address of a structure descriptor block (SDB) that identifies the semaphore to be conditionally decremented. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=] sdb-address

**Restrictions** - The semaphore identified by the passed SDB must not be a queue semaphore.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the WAICSp macro has the following format:



## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Syntax Example

```
WAIC$P Sdb=$BSPM
```

### Semantics

The WAIC\$ primitive decrements the specified semaphore variable if its current value is greater than 0 and returns to the caller with the kernel-defined value TRUE in R0. If the semaphore value is already 0, the WAIC\$ primitive returns immediately to the caller with the kernel-defined value FALSE in R0.

#### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

The WAIC\$ primitive is a decrement open (signaled) semaphore function. It returns a successful/unsuccessful indication and never causes the calling process to block.

### Error Returns

F.IID - Invalid semaphore description (index or name); no such binary or counting semaphore exists.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.43 WAIQS (WAIT ON QUEUE SEMAPHORE)

Pascal equivalent: GET\_PACKET Procedure

The Wait on Queue Semaphore (WAIQS) primitive tests the specified semaphore for an available packet. If one is available, it is removed from the semaphore's packet queue, and a pointer to the packet is returned to the caller. If no packet is available, the calling process is blocked on the semaphore, awaiting a subsequent signal.

This primitive permits the calling process to receive a signal from another process that a data packet it is dependent on is available, regardless of the order in which the signal and wait calls occur. (Compare with WAQCS, the conditional wait on queue call.)

A packet is a fixed-length system data structure allocated by the kernel from a special system-memory pool; see the ALPKS primitive and Section 2.2.2. The overall size of a packet, including the 3-word header, is given by the global symbol SE.SIZ, in bytes. The length of the undefined, arbitrarily usable portion of the packet is given by the global symbol QE.LEN, also in bytes. The value of QE.LEN is 34(decimal) for the MicroPower/Pascal system as distributed. Privileged processes can obtain an empty packet, for use in a SGLQS operation, by means of an allocate packet (ALPKS or ALPCS) request. (The packet is not initialized to zeros.) When no longer needed, a packet can be returned to the kernel -- freed for reuse -- by means of the DAPKS primitive.

General and device-access processes do not have direct access to a packet -- for example, may not fetch from one directly -- in a mapped environment. Therefore, if such a process needs to extract data from an acquired packet, as opposed to passing it along via another primitive, it must use the RCVD primitive, which provides a data-copying and packet-deletion service in addition to the functionality of WAIQS.

The SGLQS call is the inverse of the WAIQS call.

#### Syntax

The three variants of the WAIQS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
WAIQS	WAIQS [area,sdb,qelm]
WAIQSS	WAIQSS [sdb,qelm]
WAIQSP	WAIQSP [sdb]

#### Parameter      Definition

area      The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:

[AREA=]arg-blk-address

## MACRO-11 PRIMITIVE SERVICE REQUESTS

sdb

The address of a structure descriptor block (SDB) that identifies the semaphore to be waited on. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

[SDB=] sdb-address

qelm

The address of a location in which the packet address is to be returned by the primitive. This argument may be null; otherwise, it has the form:

[QELM=] destination-address

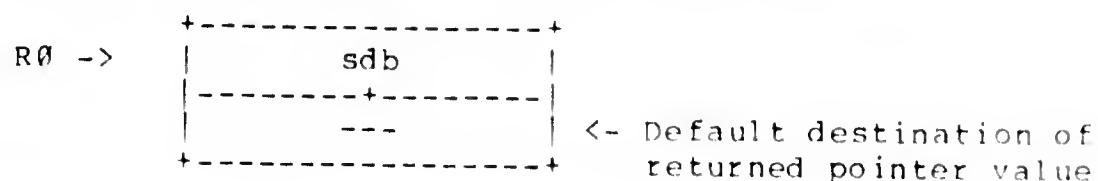
If the qelm argument is null, the packet pointer returned by the primitive is available only in the last word of the calling argument block. If the argument is null in the stack (\$S) version of the macro call, the returned pointer value is left on the stack. In the parameters-only (SP) version of the macro call, no qelm argument is specified, and the returned pointer value is available only in the last word of the calling argument block. (See the Restrictions section below.)

**Restrictions** - The semaphore identified by the passed SDB must be a queue semaphore.

The argument block must be in read/write memory.

You can use the parameters-only (SP) version of the macro call in a RAM-only system, provided that you correctly access the queue element word in the argument block. However, you cannot use the SP call in either the ROM or RAM portion of a ROM/RAM system.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the WAIQSx macro has the following format:



### Syntax Example

```
WAIQS area=#WARGS,qelm=#PKTPTR
```

### Semantics

The WAIQS primitive decrements the specified semaphore and tests for an available packet. If at least one packet is on the semaphore's packet queue, the primitive removes the first available packet from the queue and returns the address of that packet in the last word of the argument block. If requested (qelm argument), the macro expansion moves the address to a user-specified location.

If no packets are on the semaphore's packet queue, the primitive blocks the calling process and calls the scheduler. The process remains on the semaphore's waiting process list until it is unblocked by a signal of the semaphore, which places a packet on the queue.

## **MACRO-11 PRIMITIVE SERVICE REQUESTS**

### **Error Returns**

E.IIID - Invalid semaphore description (index or name); no such queue semaphore exists.

### **Applications**

See the SGLQS description.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### 3.45 WAQCS (CONDITIONAL WAIT ON QUEUE SEMAPHORE)

Pascal equivalent: COND\_GET\_PACKET Function

The Conditional Wait on Queue Semaphore (WAQCS) primitive tests the specified semaphore for an available packet. If one is available, it is removed from the semaphore's packet queue. A pointer to the packet is returned to the caller, and the kernel-defined value TRUE is returned in R0. If no packet is available, the primitive returns immediately to the caller with the kernel-defined value FALSE in R0.

This primitive permits the calling process to receive a signal from another process that a data packet is available, but without blocking on the semaphore if the signal hasn't already occurred. (Compare with WAIQS, the unconditional wait on queue call.)

See the WAIQS primitive for a description of queue packets. General and device-access processes do not have direct access to a packet -- for example, may not fetch from one directly -- in a mapped environment. Therefore, if such a process needs to extract data from an acquired packet, as opposed to passing it along via another primitive, it must use the RCVC primitive, which provides a data-copying and packet-deletion service in addition to the functionality of WAQCS. The inverse of the WAQCS call is the SGQS call.

#### Syntax

The three variants of the WAQCS macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
WAQCS	WAQCS [area,sdb,qelm]
WAQCSS	WAQCSS [sdb,qelm]
WAQCSP	WAQCSP [sdb]
Parameter	Definition
area	The address of a user-memory area in which the calling argument block is to be constructed (or found if already existent). This argument has the form:  [AREA=]arg-blk-address
sdb	The address of a structure descriptor block (SDB) that identifies the semaphore to be tested. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:  [SDB=] sdb-address
qelm	The address of a location in which the packet address is to be returned by the primitive. This argument has the form:  [QELM=]destination-address  Or, it may be null.

## MACRO-11 PRIMITIVE SERVICE REQUESTS

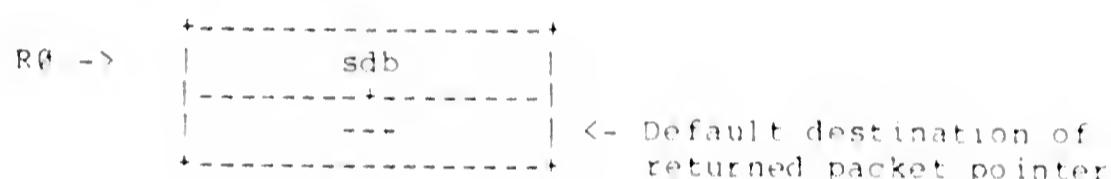
If the qelm argument is null, the packet pointer returned by the primitive is available only in the last word of the calling argument block. If the argument is null in the stack (\$S) version of the macro call, the returned pointer value is left on the stack. In the parameters-only (SP) version of the macro call, no qelm argument is specified, and the returned pointer value is available only in the last word of the calling argument block. (See the Restrictions section below.)

**Restrictions** - The semaphore identified by the passed SDB must be a queue semaphore.

The argument block must be in read/write memory.

You can use the parameters-only (SP) version of the macro call in a RAM-only system, provided that you correctly access the queue element word in the argument block. However, you cannot use the SP call in either the ROM or RAM portion of a ROM/RAM system.

**Argument Block** - The calling argument block generated -- or assumed to exist -- by the WAQC\$X macro has the following format:



### Syntax Example

```
WAQCSS  sdb=#QSEM,qelm=R3
```

### Semantics

The WAQC\$ primitive tests the specified queue semaphore for an available packet. If at least one packet is on the semaphore queue, the primitive removes the first available packet from the packet queue and returns the address of that packet in the last word of the argument block. If requested (qelm argument), the macro expansion moves to the address to a user-specified location. The primitive also returns the kernel-defined value TRUE in R0.

If no packets are on the semaphore queue, the primitive returns to the caller, with the kernel-defined value FALSE in R0.

### NOTE

TRUE and FALSE are absolute symbols defined by the kernel. In the current version of MicroPower/Pascal, the symbol TRUE has a value of 1 and the symbol FALSE has a value of 0. The following MACRO-11 instruction could be used to check for a returned value of TRUE in R0:

```
CMP #TRUE,R0
```

## MACRO-11 PRIMITIVE SERVICE REQUESTS

### Error Returns

E.IIID - Invalid semaphore description (index or name); no such queue semaphore exists.

## CHAPTER 4

### STANDARD DEVICE HANDLERS

This chapter describes the standard device-handler interface for both Pascal and MACRO-11 users. Section 4.1 describes the general features of the user/handler interface; subsequent sections specify the device-dependent aspects of the interface for each handler.

The MicroPower/Pascal standard device handlers fall into two major categories: those for use on a KXT11-C slave processor and those for use on any other PDP-11 processor. The latter are described in Sections 4.2 through 4.13; the KXT11-C handlers are described in Sections 4.14 through 4.19.

Table 4-1 lists the MicroPower/Pascal standard device handlers in the order that they are described in this chapter. Tables 4-2 and 4-3 provide a detailed list of hardware supported by those handlers.

Table 4-1  
Standard Device Handlers and Device Identifiers

Device Handler	Identifier
ADV11-C/AXV11-C Analog Input Converter	AA
TU58	DD
RL01/RL02	DL
MSCP Disk-Class Device	DU
RX02	DY
KWV11-C Real-Time Clock	KW
KXT11-C Two-Port RAM (arbiter-side)	KX
DRV11-J Parallel Line	XA
Serial Line	XL
DPV11 Communications Line	XP
DRV11 Parallel Line	YA

(Continued on next page)

## STANDARD DEVICE HANDLERS

Table 4-1 (Cont)  
Standard Device Handlers and Device Identifiers

Device Handler	Identifier
SBC-11/21 Parallel Port	YF
Line-Frequency Clock (see Chapter 6)	CK
KXT11-C TU58	DD
KXT11-C Two-Port RAM (KXT11-C side)	KK
KXT11-C DMA Transfer Controller	QD
KXT11-C Asynchronous Serial Line	XL
KXT11-C Synchronous Serial Line	XS
KXT11-C Parallel Port and Timer/Counter	YK

### 4.1 GENERAL DEVICE-HANDLER INTERFACE

A MicroPower/Pascal device handler is a process or a family of cooperating processes that accepts requests for device-level I/O operations from other processes. The handlers use standard primitive operations to communicate with requesting processes.

An application process communicates with a device handler by signaling a queue semaphore established by the handler for that purpose. (Such a semaphore is called the handler's request queue.) The requesting process sends its I/O request message to the handler via the request queue, using either the Pascal SEND procedure or the MACRO-11 SEND\$ primitive call.

The request packet supplies all the information the handler needs to perform the desired operation, including the function code, type of reply desired, and, where applicable, the unit number, physical or logical device address, and data-buffer location. Some elements of the request message are device-dependent; those elements are defined separately for each handler in later sections of this chapter.

If a full reply is requested, the handler will send back a modified version of the request message through a user-specified queue semaphore. That reply packet contains completion status and, possibly, error information, along with most of the information originally sent in the request packet.

#### NOTE

The device handler request and reply packets are described in Sections 4.1.2 and 4.1.3 and throughout the rest of this chapter. The symbols used in this chapter to describe the packets and the information they contain are MACRO-11 symbols defined by the kernel macro DRVDFS (from the DRVDEF.MAC library).

## STANDARD DEVICE HANDLERS

The Pascal equivalents of those symbols are defined in IOPKTS.PAS, an include file that is recommended for use with Pascal device-level I/O requests.

### 4.1.1 Request Queue Names

The handler request queue semaphores have standardized, 4-character names that indicate the device class, device type, and controller serviced by a handler. The names are of the form Scda:

Designator	Meaning
c	A device class
d	A device type
a	A given controller (for example, A, B, C, 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 \$XLA would name the queue for any DLV11 interface.

The request queue name must be specified in uppercase letters. Also, since device handlers 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.

The ADV11-C/AXV11-C (AA) handler, the KWV11-C (KW) handler, and the clock process do not support multiple controllers.

#### NOTE

Specific device register addresses, interrupt vector addresses, and other configuration information describing the target hardware must be made available to the appropriate device handlers. You do that while building the application system by editing a device-handler prefix file that describes the target system hardware. Both the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide give a detailed procedure for editing and subsequently processing that file.

Table 4-2 lists standard device queue names, device-handler prefix files, and supported hardware units for the PDP-11 (non-KXT11-C) standard device handlers. Table 4-3 lists the same information for the KXT11-C standard device handlers.

## STANDARD DEVICE HANDLERS

Table 4-2  
PDP-11 Request Queue Names, Prefix Files, and Hardware

Device Queue Name*	Device-Handler Prefix File	Units per Controller	Hardware Supported
\$AAA	AAPEX.PAS	1	ADV11-C, AXV11-C
\$DDA	DDPEX.MAC	1-2	TU58 (DL11-type controller)
\$DLA	DLPFX.MAC	1-4	RL01, RL02 (RLV11, RLV12, and RLV21 controllers)
\$DUA	DUPFX.MAC	1-4	MSCP disk-class devices, including RD51 and RX50 (RQDX1 controller)
\$DYA	DYPFX.MAC	1-2	RX02 (RXV21 controller)
\$KWA	KWPEX.PAS	1	KWV11-C
\$KXA	KXPEX.MAC	1-2	KXT11-C Two-Port RAM (arbiter-side)
\$XAA	XAPFX.MAC	1-4	DRV11-J
\$XLA	XLPEX.MAC	1-4	DLV11, DLV11-E, -F, -J, MXV11-A, MXV11-B, SBC-11/21 (serial line)
\$XPA	XPPFX.MAC	1	DPV11
\$YAA	YAPFX.PAS	1	DRV11
\$YFA	YFPFX.MAC	1-2	SBC-11/21 (PIO port only)
SCLOCK	CKPEX.MAC	(none)	Clock process (see Chapter 6)

\* Device queue names are shown only for one controller (A). When more than one controller is used, subsequent controllers are designated B, C, and so forth. For example, a second TU58 controller would have an associated device queue named \$DDB.

Table 4-3  
KXT11-C Request Queue Names, Prefix Files, and Hardware

Device Queue Name*	Device-Handler Prefix File	Units per Controller	Hardware Supported
\$DDA	DDPEFK.MAC	1-2	TU58
\$KKA	KKPEFK.MAC	1-2	KXT11-C Two-Port RAM (KXT11-C side)
\$QDA	QDPEFK.MAC	1-2	KXT11-C DMA Transfer Controller
\$XLA	XLPEFK.MAC	1-3	KXT11-C asynchronous serial line

(Continued on next page)

## STANDARD DEVICE HANDLERS

Table 4-3 (Cont)  
KXT11-C Request Queue Names, Prefix Files, and Hardware

Device Queue Name*	Device-Handler Prefix File	Units per Controller	Hardware Supported
\$XSA	XSPFXK.PAS	1	KXT11-C synchronous serial line
SYKA	YKPEFK.MAC	1-6	KXT11-C parallel port and timer/counter
\$CLOCK	CKPEFX.MAC	(none)	Clock process (see Chapter 6)

\* Device queue names are shown only for one controller (A). When more than one controller is used, subsequent controllers are designated B, C, and so forth.

### 4.1.2 I/O Request Packet

Figure 4-1 shows the general form of an I/O request packet as received by the handler. The diagram excludes the standard 3-word header that prefixes all packets and that is transparent to users of the send/receive-level mechanisms.

Note that the request data consists of a 9-word portion that is function-independent -- fields DP.FUN to DP.SEM -- and a 3- to 5-word portion that varies in form, depending on the kind of function requested. The format of the function-dependent portion shown in the diagram, beginning with field DP.DAD, is representative for read/write functions. (The field names used at the left of the diagram correspond to the offset symbols used by the handlers; for example, DP.FUN is a 6-byte offset from the packet header.)

## STANDARD DEVICE HANDLERS

DP.FUN -	Function	
DP.UNI -	-----  Unit	
DPSEQ -	Seq. number	
DP.PDB -	Requestor's -- process -- identifier	Function-independent portion of request
DP.SEM -	Reply -- semaphore -- identifier	
* DP.DAD -	Device -- address	Function-dependent portion of request (format for a read or write function is illustrated)
DP.BUF -	Buffer address	
DP.PAR -	PAR value	<- Supplied by SEND/SENDS
DP.LEN -	Buffer length	

\* This field, the beginning of the function-dependent portion, is, alternatively, a 3-word ring buffer or semaphore identifier, designated as DP.SGL, for certain handler requests.

Figure 4-1 I/O Request Packet Format

For random-access devices, such as the RX02, the device address field (DP.DAD) has two different formats, depending on whether the type of function requested is physical or logical. The two formats are as follows:

### Physical Address Format

DP.DAD -	-----+-----  Track   Sector
	--  Cylinder

### Logical Address Format

DP.DAD -	-----+-----  Number of	(Least significant)
	--  logical block	(Most significant)

## STANDARD DEVICE HANDLERS

The TUS8, although a random-access device, uses one DP.DAD format for both physical and logical function requests.

The request packet fields shown in Figure 4-1 have the following significance:

Field	Significance
DP.FUN	<p>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:</p> <ul style="list-style-type: none"><li>• Function code value in bits 0 to 5: IFSPPD = Read physical IFSRDL = Read logical IFSWTP = Write physical IFSWTL = Write logical IFSSFT = Set device characteristics IF\$GET = Get device characteristics</li><li>Other codes denote device-specific functions; for example, IFSCRR, IFSERR, as used in the XL handler requests. (Refer to individual device-handler descriptions for device-specific functions.)</li><li>• Device-dependent function-modifier bits 6 to 12 (their meaning is described separately for each handler).</li><li>• Device-independent function-modifier bit settings, for bits 13 to 15: FMSBSM (bit 13) Set = Reply semaphore (DP.SEM) is a binary or a counting semaphore FMSDCR (bit 14) Set = Data check FMSINH (bit 15) Set = Inhibit retry of soft device errors</li></ul>
DP.UNIT	The unit number of the desired device, where applicable. (The high-order byte of DP.UNIT is reserved.)
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 handler 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).
DP.SEM	The Pascal STRUCTURE_ID-type variable that identifies the user's completion-reply semaphore (first three

## STANDARD DEVICE HANDLERS

words of the structure descriptor block; see Section 3.1.5). If modifier bit FMSBSM 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 FMSBSM 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 handler takes no completion-reply action.

DP.DAD	Interpreted according to the type of device handler and the function requested and is unused in some cases. For physical I/O on a disk device, the two words are interpreted as a physical device address; for logical I/O on a disk, the two words are interpreted as a LONGINT-type (double-word) logical block number, with the least-significant part in the first word. For the TUS8, the first word is interpreted as a block number for both physical and logical function requests. Other handlers either ignore this field or interpret it as a 3-word structure identifier (for example, as a ring buffer for serial line I/O). This field is known as DP.SGL when used as a structure identifier.
DP.BUF	The virtual address of the start of the user's data buffer. This word is filled in automatically by SEND or SENDS, 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 SENDS 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 SENDS, based on the reference-length parameter you supply in the call.

For a Set Characteristics function, the function-dependent portion of the request would contain one or more words of device characteristics data, beginning at DP.DAD.

Note that all handlers will notify the requesting process of a request completion, if a reply semaphore was specified in the request (DP.SEM is nonzero), by either a full reply or a lone signal, as determined by function-modifier bit FMSBSM of the function word (DP.FUN). If bit FMSBSM is not set, a full reply is sent via the queue semaphore identified in DP.SEM; the format of the reply packet is given in Section 4.1.3.

If bit FMSBSM 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.

## STANDARD DEVICE HANDLERS

4.1.2.1 Sample I/O Request in Pascal - The MicroPower/Pascal program fragments given below illustrate the following:

- A record structure that might be used for building an RX02 logical-I/O request message
- The form of SEND procedure call for transmitting the request to the appropriate handler

The RX02 handler's request queue name is SDYA, assuming that controller A is desired.

### NOTE

DIGITAL supplies an include file, IOPKTS.PAS, that is recommended for use with Pascal I/O requests. IOPKTS defines device class, type, and subtype codes; error and severity codes; function codes; function modifier bits; and request and reply packets. The SEND call example below does not use the IOPKTS include file.

```
CONST
  max_funct_code = 63;
  read_logical = 1;
TYPE
  funct_code = 0..max_funct_code;
  fun_indep_bits = (simple_reply, data_chk, noretry);
  fun_dep_bits = (bit0, bit1, bit2, bit3, bit4, bit5, bit6);
  block = ARRAY[0..255] OF UNSIGNED;

{Device address is a two word record, with variant
definitions for physical and logical operations,
as defined below: }

device_address =
  PACKED RECORD
    CASE funct_code OF
      read_physical,
      write_physical : (sector, track : BYTE;
                        cylinder : UNSIGNED);
      read_logical,
      write_logical : (block_num : LONGINT);
    END;
{IO request is an 11 word record defined as follows: }

io_req =
  PACKED RECORD
    oper : [POS(0),BIT(6)] funct_code; { Operation }
    dep_mod : [POS(6),BIT(7)] PACKED SET OF fun_dep_bits;
    ind_mod : [POS(13),BIT(3)] PACKED SET OF fun_indep_bits;
    unit_num,                      { Unit number }
    filler_i : BYTE;               { Reserved }
    sequence : INTEGER;           { User specified sequence number }
    CASE funct_code OF
      read_physical,      { All have the same variant }
      read_logical,
      write_physical,
      write_logical :
        (pid,                { Requestor's ID }
         reply_sem : STRUCTURE_ID; { Reply semaphore ID }
         dev_addr : device_address);
    END;
```

## STANDARD DEVICE HANDLERS

```
io_reply =
PACKED RECORD
  oper : [POS(0),BIT(5)] funct_code; { operation }
  dep_mod : [POS(13),BIT(7)] PACKED SET OF fun_dep_bits;
  ind_mod : [POS(13),BIT(3)] PACKED SET OF fun_ind_bits;
  unit_num,           { Unit number }
  filler_1 : BYTE;      { Reserved }
  sequence : UNSIGNED;   { User specified sequence number }
  status,
  actual_length,
  err_code : INTEGER;
END;
VAR
  IO_buffer: block;
  Read_reply, Device_driver : QUEUE_SEMAPHORE_DESC;
  Read_io_request : To_req;
  Read_io_reply : io_reply;
  Proc_id : PROCESS_DESC;
  Block_num, Seq_num : INTEGER;
  .
  .
  .
INIT_PROCESS_DESC( DESC := Proc_id ); {Identifies the requesting
process.}

INIT_STRUCTURE_DESC( DESC := Device_driver,
NAME := 'SDYA' );
{The SDYA controller request semaphore should already exist--  
having been created by DY driver initialization. }

IF NOT CREATE_QUEUE_SEMAPHORE( DESC := Read_reply )
then
  { Report error and don't proceed to following statements since  
we want to have a reply semaphore with which the DY driver will  
report status on completion of the operation. }
  .
  .
  .
{ Following assumes that Block_num and Seq_num have been  
established and that disk unit number to read from  
is 1 (that is, DY1 is the disk device). }

WITH Read_io_request DO
BEGIN
  oper := read_logical;
  ind_mod := [1];
  dep_mod := [1];
  sequence := Seq_num;
  reply_sem := Read_reply::STRUCTURE_ID;
  pid.index := Proc_id.index;
  pid.serial_number.high := Proc_id.serial_number.high;
  pid.serial_number.low := Proc_id.serial_number.low;
  dev_addr.block_num.high := 0;
  dev_addr.block_num.low := Block_num;
  unit_num := 1;
END;
.
.
{ Send request to the handler's request queue for controller A. }
SEND ( REF_DATA := IO_buffer,
REF_LENGTH := SIZE( IO_buffer ),
PRIORITY := 20,          {Optional priority}
VAL_DATA := Read_io_request,
VAL_LENGTH := SIZE( Read_io_request ),
DESC := Device_driver );
.
.
{ Wait for reply from handler. See Section 4.1.3 for sample reply. }
```

## STANDARD DEVICE HANDLERS

Note that the SEND call above transmits 11 words of request data by value and 3 words by reference (the I/O buffer address and length, as well as a PAR value supplied by the send operation).

**4.1.2.2 Sending an I/O Request in MACRO-11** - Read or write requests normally include the I/O buffer address and length fields (DP.BUF and DP.LEN) shown in Figure 4-1. (An exception is requests for data transfers to or from a ring buffer.) To send that kind of request using the SEND\$ primitive, the portion DP.FUN to DP.UNI of the request must be sent by value, and the DP.BUF and DP.LEN values must be sent by reference, as follows:

```
SEND$ area,sdb,pri,#22.,vbuf,rlen,rbuf
```

In the primitive above, vbuf points to the caller's 22-byte send-by-value buffer, and rbuf and rlen specify the DP.BUF and DP.LEN values, respectively. The format of the caller's send buffer is shown in Figure 4-2.

Note that the SEND\$ primitive transforms the pair of values specified by rptr and rlen into three words in the packet received by the handler. SEND\$ does that by inserting a user PAR value for the I/O buffer address (that is, field DP.PAR).

vbuf ->	Function	(DP.FUN)
	0   Unit	(DP.UNI)
	Seq. number	(DPSEQ)
	Requestor's -- process	(DP.PDB)
	descriptor	
	Reply	(DP.SEM)
	semaphore	
	descriptor	
	Device addr. -- or -- block number	(DP.DAD)

Figure 4-2 Format of Send Buffer for an I/O Request

## STANDARD DEVICE HANDLERS

### NOTE

The send buffer for an I/O request should not include the 3-word (6-byte) packet header for the packet to be constructed. That header will be provided by the SENDS primitive, based on kernel information and user-provided macro arguments, when it builds the packet. (See Section 3.33 for a detailed description of the SENDS primitive.)

The field names shown in Figure 4-2 do not represent offsets into the user's send buffer. Rather, they correspond to offset symbols used by the handler to reference packets; for example, DP.FUN is a 6-byte offset from the packet header.

To send other than block-mode read/write requests -- requests that do not require an I/O buffer address -- all the request data is sent by value, as follows:

```
SENDS area,sdb,pri,#24.,vbuf,#0,#0
```

This form of SENDS call would copy 24 bytes of request data from the caller's send buffer (vbuf) to the request packet. The format of the first nine words (fields DP.FUN to DP.SEM) of the send buffer is the same as that shown in Figure 4-2. The format of the function-dependent portion following field DP.SEM will vary, depending on the type of function requested and the kind of device involved. The function-dependent portion of a request is described in detail for each handler in the individual handler descriptions.

#### 4.1.3 Completion Reply Format

If a full reply was requested as described in Section 4.1.2, the handler sends a reply packet to the user-specified reply queue semaphore. The reply message is a modified form of the request message, with all fields unchanged except for DP.PDB, in which completion-status information has been inserted and, possibly, DP.SEM.

The general format of a reply message, as received by the requestor via the RECEIVE procedure or RCVDS primitive, is shown in Figure 4-3. The length of a given reply message depends on the size of the function-dependent portion of the corresponding request. (You need not receive all the data contained in the reply packet, of course.)

## STANDARD DEVICE HANDLERS

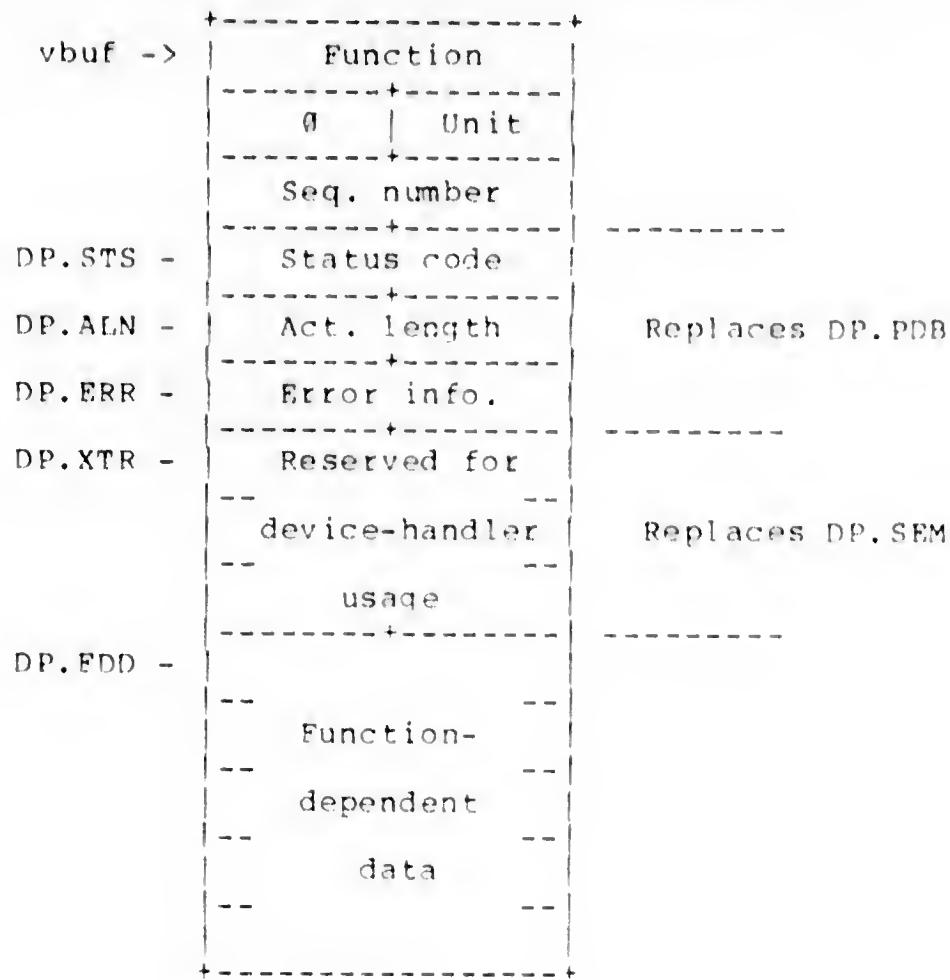


Figure 4-3 Format of an I/O Reply Message

### NOTE

An I/O reply message does not include the 3-word (6-byte) header of the reply packet; accordingly, the reply buffer specified in the RECEIVE or RCVDS call should not include the 3-word header. (The RECEIVE and RCVDS calls are shown in Sections 4.1.3.1 and 4.1.3.2.)

The field names shown in Figure 4-3 do not represent offsets into the user's reply buffer. Rather, they correspond to offset symbols used by the handler to reference packets. For example, DP\_STS is a 12-byte offset from the packet header.

The meanings of the modified fields in the reply message shown in Figure 4-3 are as follows:

Field	Significance	
DP_STS	Code for completion status, indicating error (if any) and severity level, as follows:	
Code	Error	Severity
ISSNOR	Normal operation	Success
ISSABT	I/O abort	Fatal

Code	Error	Severity
ISSNOR	Normal operation	Success
ISSABT	I/O abort	Fatal

## STANDARD DEVICE HANDLERS

ISSATN	Attention	Error
ISSCTL	Controller error	Fatal
ISSDAL	Device allocated	Error
ISSDRV	Drive error	Error
ISSEOF	End of file on device	Error
ISSEOV	End of volume	Error
ISSFOR	Format error	Error
ISSFUN	Invalid I/O function code	Error
ISSIBN	Invalid block number	Error
ISSIDA	Invalid device address	Error
ISSIVM	Invalid mode	Error
ISSNXM	Nonexistent or read-only memory	Error
ISSNXU	Nonexistent unit	Error
ISSOFL	Device off line or not mounted	Error
ISSPAR	Bad parity	Fatal
ISSPNA	Packet not available	Error
ISSPWR	Power fail	Error
ISSTIM	Device time out	Error
ISSUNS	Unsafe volume	Error
ISSVOL	Volume invalid	Error
ISSWLK	Write-locked unit	Error
ISSSTP	I/O processing stopped	Info
ISSOVF	Overflow	Error
ISSOVR	Overrun	Error
ISSIVP	Invalid parameter	Error
ISSIVD	Invalid data	Error

The IOPKTS.PAS include file defines the Pascal equivalents of the MACRO-II status codes shown above (see the IOSSTATUS, IOSSEVERITY, and IOSError data types). They are as follows:

MACRO-II	Pascal
ISSNOR	IOSSTATUS(ISSUCCESS, IESNORMAL)
ISSABT	IOSSTATUS(ISSFATAL, IESABORT)
ISSATN	IOSSTATUS(ISSERROR, IESATTENTION)
ISSCTL	IOSSTATUS(ISSFATAL, IESCNTRLR)
ISSDAL	IOSSTATUS(ISSERROR, IESDEVALLOC)
ISSDRV	IOSSTATUS(ISSERROR, IESDRIVE)
ISSEOF	IOSSTATUS(ISSERROR, IESENDOFFILE)
ISSEOV	IOSSTATUS(ISSERROR, IESENDOFVOL)
ISSFOR	IOSSTATUS(ISSERROR, IESFORMAT)
ISSFUN	IOSSTATUS(ISSERROR, IESFUNINV)
ISSIBN	IOSSTATUS(ISSERROR, IESBLKNOINV)
ISSIDA	IOSSTATUS(ISSERROR, IESDAVADRINV)
ISSIVM	IOSSTATUS(ISSERROR, IESMODEINV)
ISSNXM	IOSSTATUS(ISSERROR, IESNONEXMEM)
ISSNXU	IOSSTATUS(ISSERROR, IESNONEXUNIT)
ISSOFL	IOSSTATUS(ISSERROR, IESDEVOFFLIN)
ISSPAR	IOSSTATUS(ISSFATAL, IESPARTY)
ISSPNA	IOSSTATUS(ISSERROR, IESNOPACKET)
ISSPWR	IOSSTATUS(ISSERROR, IESPWRFAIL)
ISSTIM	IOSSTATUS(ISSERROR, IESDEVTIMOUT)
ISSUNS	IOSSTATUS(ISSERROR, IESUNSAFEVOL)
ISSVOL	IOSSTATUS(ISSERROR, IESVOLUMEINV)
ISSWLK	IOSSTATUS(ISSERROR, IESWRITELOCK)
ISSSTP	IOSSTATUS(ISSINFO, IESSTOP)
ISSOVF	IOSSTATUS(ISSERROR, IESOVERFLOW)
ISSOVR	IOSSTATUS(ISSERROR, IESOVERRUN)
ISSIVP	IOSSTATUS(ISSERROR, IESINVPARAM)
ISSIVD	IOSSTATUS(ISSERROR, IESINVDATA)

## STANDARD DEVICE HANDLERS

DP.ALEN	The length of the data actually transferred, in bytes, for I/O functions.
DP.FRR	device-dependent hardware or software error information if DP.STS is nonzero

### NOTE

Individual handler descriptions later in this chapter provide more specific information about the status, length, and error word values.

**4.1.3.1 Receiving an I/O Reply Message in Pascal** - The following material builds on the example of an RX02 logical-I/O request given in Section 4.1.2.1. The MicroPower Pascal program fragments given below illustrate the following:

- A record structure that might be used to map the reply from an RX02 logical-I/O operation
- The corresponding RECEIVE procedure call used to wait for and receive the reply message from the appropriate handler

The example assumes that the requestor wants to receive only the top part of the reply message. That portion includes the status and error information -- fields DP.SUN to DP.FRR.

### NOTE

DIGITAL supplies an include file, IOPKTS.PAS, that is recommended for use with Pascal I/O requests. IOPKTS defines device class, type, and subtype codes; error and severity codes; function codes; function modifier bits; and request and reply packets. The RECEIVE call example below does not use the IOPKTS include file.

{ Note: Value data returned to requestor after completion of driver function contains the original fields from the request packet with the three words of process descriptor replaced by status, actual length (of data transferred in bytes) and err\_code. }

### TYPE

```
in_reply =  
PACKED RECORD  
    oper : (POS(2),BIT(6)); funct_code; { operation }  
    dep_mod : (POS(6),BIT(7)); PACKED SET OF fun_dep_bits;  
    ind_mod : (POS(12),BIT(3)); PACKED SET OF fun_ind_bits;  
    unit_num, { Unit number }  
    filler_1 : BYTE; { Reserved }  
    sequence : UNSIGNED; { User specified sequence  
    number }
```

## STANDARD DEVICE HANDLERS

```
    status,
    actual_length,
    err_code : INTEGER;
END;

VAR
  Read_reply : QUEUE_SEMAPHORE_DESC;
  Read_io_reply : io_reply;
  .
  .
  .
  RECEIVE ( VAL_DATA := Read_io_reply,
             VAL_LENGTH := SIZE(Read_io_reply),
             DESC := Read_reply );
  IF Read_io_reply.status <> 0
  then
    { Operation was not completed successfully;
      therefore, data in IO buffer is not good
      and appropriate action should be taken. }
  else
    { Operation was successful; data in IO-buffer
      is good; proceed accordingly.}
  .
  .
  .
```

The user's reply queue semaphore is identified by the descriptor variable QUEUE\_SEMAPHORE\_DESC, which was declared and initialized by the program fragment shown in Section 4. 2.1.

**4.1.3.2 Receiving an I/O Reply Message in MACRO-11** - The MACRO-11 equivalent to the Pascal RECEIVE call is the RCVDS primitive call. That primitive can be used to receive a maximum-length reply message, as follows:

```
RCVDS area,sdb,rtnptr,#28.,vbuf,#0,#0
```

The receive-length parameter, shown above as 28 bytes, may, of course, be a lesser value reflecting the amount of reply data that the requestor wishes to receive for a given function. Note that only the first 12 bytes of a reply need be received in order to obtain completion status and error information.

## STANDARD DEVICE HANDLERS

### 4.2 ADV11-C/AXV11-C (AA) ANALOG INPUT CONVERTER HANDLER

The AA device handler supports analog-to-digital (A/D) conversions on both the ADV11-C and the AXV11-C analog-to-digital converter boards. The handler also reports standard device characteristics.

The ADV11-C and the AXV11-C analog-to-digital converter boards each accept up to 16 single-ended analog inputs or 8 differential analog inputs. Each board has a programmable gain on these inputs of one, two, four, or eight times the input voltage.

Analog-to-digital conversions are started by a program command, an external trigger, or a real-time clock input. Converted data is copied to a block-mode buffer or a record-mode ring buffer specified by the user. Requests for A/D conversions specify the method of triggering the conversion, the number of analog input channels to sample, which channels to sample, the gain on each channel, and where to put the converted data.

#### NOTE

In addition to the 16 single-ended or 8 differential analog input channels mentioned above, the AXV11-C board has two digital-to-analog converters (DACs). However, the digital-to-analog features of the AXV11-C are not supported by the AA device handler. Instead, digital-to-analog conversions are supported by the DIGITAL-supplied procedure `Write_analog_wait`, which uses programmed I/O and requires I/O page access. See Appendix G for a detailed description of the `Write_analog_wait` procedure.

The AA handler consists of a request-handling static process and several routines, including separate block-mode and ring buffer-mode interrupt service routines (ISRs). The AA handler does not support multiple controllers; therefore, only one AA handler static process can be resident in a target system. The AA handler process has a request queue semaphore on which it waits for I/O requests. The handler's request queue semaphore name is SAAA. The handler process also maintains an internal read-request queue to handle overlapping requests.

Each ADV11-C or AXV11-C board is factory configured with a standard device address and a standard interrupt vector address. If you do not want standard address assignments, you must reconfigure the AA controller 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 configuration file that is edited and processed during the building of the application software. (See the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide for software configuration procedures.)

## STANDARD DEVICE HANDLERS

### NOTE

As an alternative to the standard AA handler interface described in this section, DIGITAL supplies three MicroPower/Pascal procedures that support the analog-to-digital features of the ADV11-C and the AXV11-C: Read\_analog\_wait, Read\_analog\_signal, and Read\_analog\_continuous. See Appendix G for detailed descriptions of these procedures.

#### 4.2.1 Functions Provided

The AA handler functions provided are listed below, by symbolic and decimal function code:

Code	Function Performed
IF\$GET (7)	Get Characteristics
IF\$CRR (8)	Connect Receive Ring Buffer
IF\$STP (9)	Stop
IF\$RDC (12)	Read Converted Data
IF\$CRD (13)	Continuous Read

**4.2.1.1 Get Characteristics Function** - The IF\$GET function returns, in the function-dependent portion of the reply message, bit settings that indicate the device class and the device type.

**4.2.1.2 Connect Receive Ring Buffer Function** - The IF\$CRR function connects a user-specified ring buffer to the analog-to-digital converter. No data transfers are initiated by this request. This request is followed by the IF\$CRD request, which initiates the data transfers.

**4.2.1.3 Stop Function** - The IF\$STP function cancels pending I/O requests and notifies the requestor of their cancellation.

## STANDARD DEVICE HANDLERS

**4.2.1.4 Read Converted Data Function** - The IF\$RDC function is a physical I/O operation. You can specify to the handler the number of channels to sample, which channels to sample, the gain on each channel, the method of triggering the initial sample, and where to put the converted data. The Read Converted Data operation differs from a read physical operation in that a pointer to a special control data structure is passed in addition to the pointer to a sequential buffer. This special data structure contains the following control information for the handler: the number of channels to sample each period, the channel numbers to sample, and the gain on each channel.

**4.2.1.5 Continuous Read Function** - The IF\$CRD function initiates the data transfer through a ring buffer. The ring buffer must have been specified in a preceding IF\$CRR request. At each sample point, a conversion status word plus 1 to 16 digitized values are put into the specified ring buffer.

**4.2.1.6 Device-Dependent Function Modifiers** - The AA handler recognizes two device-dependent function modifier bits of the function word (DP.FUN): FMSRTC and FMSEXT. These function modifier bits are used only with the Read Converted Data function and are discussed in Section 4.2.2.4.

**4.2.1.7 Device-Independent Function Modifiers** - If modifier bit FMSRSM of DP.FUN is set, the AA handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3.

### 4.2.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of an AA handler request, following field DP.SEM, is described below for each type of function.

**4.2.2.1 Get Characteristics Function** - The Get Characteristics request (IF\$GET) returns the following information in the function-dependent portion of the reply packet:

DP.FDD	Type	Class

In the information above:

- Class is DC\$RLT, for real-time device class
- Type is RTSADV for the ADV11-C, RTSAXV for the AXV11-C

## STANDARD DEVICE HANDLERS

**4.2.2.2 Connect Receive Ring Buffer Function** - You must specify record-mode input and output when creating the ring buffer to be connected to the analog-to-digital converter. The ring buffer must be at least twice the size of a single record. A record consists of a status word plus 1 to 16 digitized values, as specified in the user request. For example, if you are requesting samples of four channels of data, the record size is 5 words (10 bytes), and the ring buffer size should be a multiple of 10 bytes that is greater than or equal to 20 bytes. Note that if the ring buffer overflows, data is lost.

No input operations are initiated by this request. Input is initiated by the continuous read request. (See Section 4.2.2.5.)

The function-dependent portion of a ring buffer connect request is shown below:

DP.RBF -	-----+-----    Ring buffer    -----+-----   -- Structure --   -----+-----    ID    -----+-----    Ignored     if     present    -----+-----	Portion of the request sent by value
----------	---	---

**4.2.2.3 Stop Function** - The function-dependent portion of a stop function request (IFSSTP) is ignored by the handler and may be omitted for this function.

**4.2.2.4 Read Converted Data Function** - The function-dependent portion of a Read Converted Data request (IFSRDC) is shown below:

DP.IPB -	-----+-----    Parameter address    -----+-----   -- PAR value --   -----+-----	Portion of the request sent by value
DP.BUF -	-----+-----    Buffer address    -----+-----	-----
DP.PAR -	-----+-----    PAR value    -----+-----	Portion of the request sent by reference

## STANDARD DEVICE HANDLERS

Two function-dependent modifier bits of the function word (DP.FUN) are used with this request only: FM\$RTC and FM\$EXT. If modifier bit FM\$RTC is set to 1 in a Read Converted Data operation, the analog conversion of the first specified channel is initiated by the real-time clock. This assumes that the real-time clock (KVV11-C) is present and that the clock output signal (CLK OV) -- clock overflow, asserted low) is connected to RTC IN B (real-time clock input, asserted low) on the A/D board.

If modifier bit FM\$EXT is set to 1 in a Read Converted Data operation, the analog conversion of the first specified channel is initiated by the external event trigger input. This assumes either that an external event signal, asserted low, is connected to EXT IN B at the I/O connector on the board and jumper F2 is connected or that jumper F1 is connected for the LSI-11 bus BEVNT line.

If neither FM\$RTC nor FM\$EXT is set, the analog conversion is initiated immediately by the AA handler. In either case, the conversions for the second to nth channels, if specified, are initiated by the device handler.

The I/O parameter address, which must be on a word boundary, specifies the location of the data structure that contains the control information for the conversion of the digitized data.

### NOTE

To assist you in generating the required physical address for the I/O parameter address specification, DIGITAL supplies a MicroPower/Pascal function -- Map\_virtual\_physical. The object module for this function, Smapvp, is contained in RHSLIB.OBJ, the analog I/O object library. The syntax for externally declaring this function is:

```
[EXTERNAL(Smapvp)] FUNCTION
    map_virtual_physical
        (virtual_addr : UNIVERSAL) :
            PHYSICAL_ADDRESS;
    EXTERNAL;
```

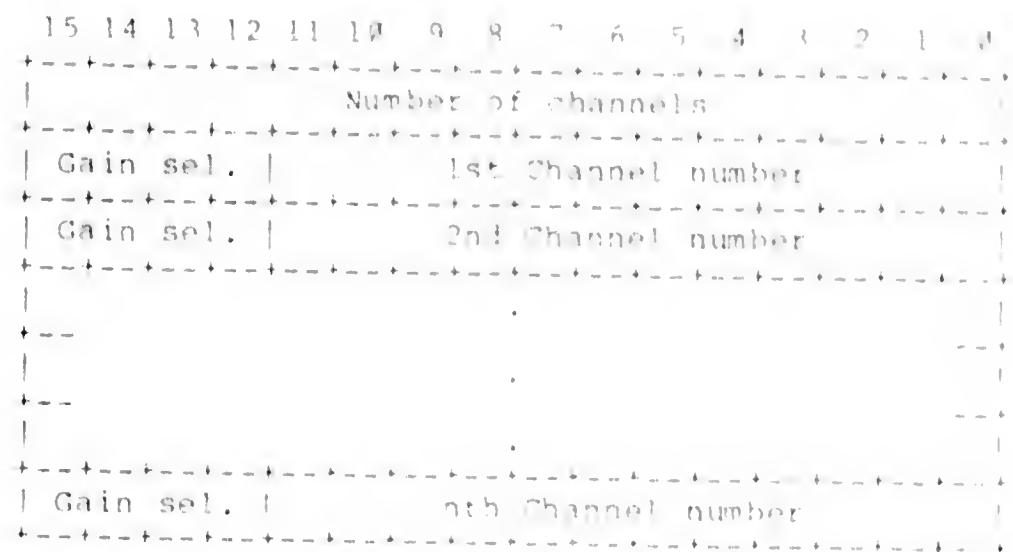
The syntax for invoking this function is:

```
... Map_virtual_physical
    (virtual_addr := address) ...
```

In this syntax, address is a variable of type UNIVERSAL, specifying the virtual address to be mapped. Map\_virtual\_physical returns a 2-word physical address that can be copied directly into the DP.IPR parameter of the request packet.

## STANDARD DEVICE HANDLERS

The I/O parameter block, which must be on a word boundary, consists of a word containing the number of channels to be sampled, followed by a channel descriptor word for each channel to be sampled. The structure size, in bytes, is twice the number of channels to be sampled, plus two. The format of the I/O parameter block is shown below.



You preset the channel descriptor words to the required values for gain and channel number. The AA device handler does not modify the channel descriptor word.

The gain select is a 4-bit unsigned binary integer, the ninth-order two bits of which are not used for the ADV11- $\mu$  and the AXV11- $\mu$ . The gain corresponding to the gain select value specified in this parameter are shown below:

Value	Gain	Range
0	1	1.0 7
1	2	5.0 7
2	4	2.5 7
3	9	1.25 11

The channel number is a 12-bit unsigned integer designating the analog input channel (multiplexer address) to be sampled. The high-order eight or nine bits of the channel number are not used. For the ADV11-C and the AXV11-C, the three or four low-order bits of the channel number select either 1 of 16 single-ended analog input channels or 1 of 8 differential input channels. Whether the analog input is single-ended or differential is determined by the type of jumper installed (S1/DP).

## STANDARD DEVICE HANDLERS

The converted data value is the result of the analog-to-digital conversion on the specified channel and gain. The range of the converted data values depends on how the jumpers are configured on the boards. The range of values may be from -4000(octal) to 3777(octal) or from 0 to 7777(octal). See the LSI-11 Analog System User's Guide, Sections 2.6.4 and 4.6.4, for details on setting the jumpers. The converted data is stored in the specified buffer, which must begin on a word boundary, as follows:

```

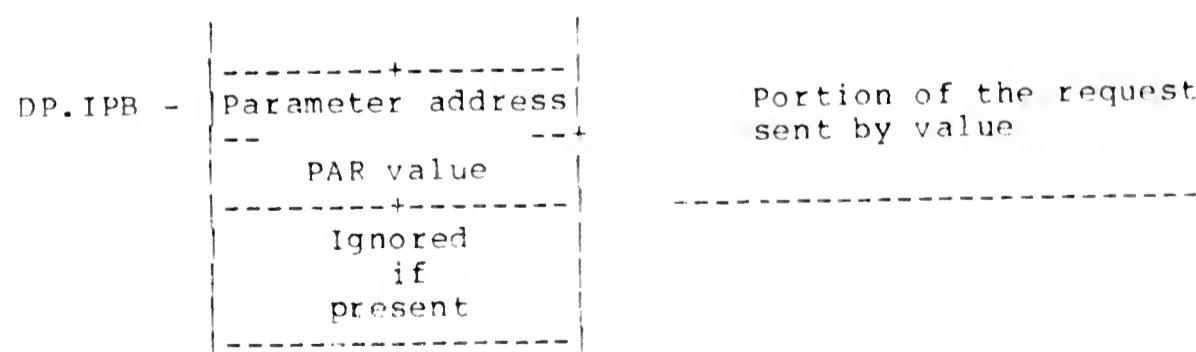
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               1st Sample, 1st Channel
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               1st Sample, 2nd Channel
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               .
|                               .
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               1st Sample, n'th Channel
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               2nd Sample, 1st Channel
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               2nd Sample, 2nd Channel
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               .
|                               .
+---+---+---+---+---+---+---+---+---+---+---+---+
|                               mth Sample, nth Channel
+---+---+---+---+---+---+---+---+---+---+---+---+

```

The buffer size should be equal to the number of sample points ( $m$ ) times the number of channels ( $n$ ) times the number of bytes per word (2), plus the number of bytes in the channel count word (2).

**4.2.2.5 Continuous Read Function** - In a continuous read operation, the analog conversion of the first specified channel is initiated by the real-time clock or an external event. The subsequent channels, as specified in the I/O parameter block, are read by programmed I/O transfer at fork level and are copied into a local buffer. When the specified channels have been read, the status for the conversion of the current sample plus the values in the local buffer are put into the ring buffer specified by the connect receive ring buffer request.

The function-dependent portion of a continuous read request (IFSCRD) is shown below:



## STANDARD DEVICE HANDLERS

The I/O parameter address, which must be on a word boundary, specifies the location of the data structure that contains the control information for the conversion of the digitized data.

### NOTE

To assist you in generating the required physical address for the I/O parameter address specification, DIGITAL supplies a MicroPower/Pascal function -- `Map_virtual_physical`. The object module for this function, `Smapvp`, is contained in `RHSLIB.OBJ`, the analog I/O object library. The syntax for externally declaring this function is:

```
[EXTERNAL(Smapvp)] FUNCTION
  map_virtual_physical
    (virtual_addr : UNIVERSAL) :
      PHYSICAL_ADDRESS;
  EXTERNAL;
```

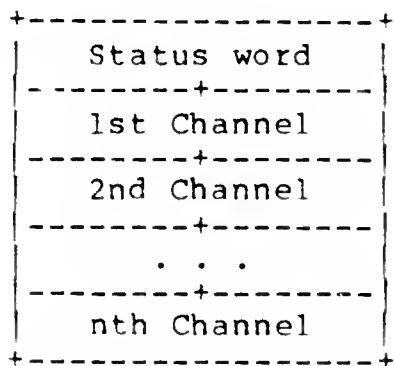
The syntax for invoking this function is:

```
... Map_virtual_physical
  (virtual_addr := address) ...
```

In this syntax, `address` is a variable of type `UNIVERSAL`, specifying the virtual address to be mapped. `Map_virtual_physical` returns a 2-word physical address that can be copied directly into the `DP.IPB` parameter of the request packet.

The I/O parameter block for a continuous read operation is identical to that used for a Read Converted Data operation. See Section 4.2.2.4 for details.

For each sampling of 1 to 16 channels, a conversion status word plus 1 to 16 words of converted data are copied to the specified ring buffer, as follows:



## STANDARD DEVICE HANDLERS

### 4.2.3 Status Codes

The AA handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal completion
ISSABT	Handler process deleted; request not serviced
ISSFUN	Invalid function code
ISSIVD	Invalid data; voltage level exceeded the range of the A/D converter at the specified gain
ISSIVM	Invalid mode
ISSSTP	Request was canceled
ISSOVR	Overrun occurred; the sampling rate requested exceeded the ability of the hardware to handle it

### 4.2.4 Extended Error Information

In the event of a hardware error, the contents of the hardware command status register (CSR) are copied into the reply packet, beginning at word DP.FDD. This information consists of two bytes of status. See the LSI-11 Analog System User's Guide for details on the contents of the hardware CSR.

## STANDARD DEVICE HANDLERS

### 4.3 TU58 (DD) DEVICE HANDLER

The DD device handler supports both logical and physical I/O operations on a TU58 cartridge tape subsystem. The handler also supports read-with-increased-threshold and write-verify options and reports the storage capacity of the device in terms of logical blocks.

Requests for logical read/write functions specify the initial tape address in terms of a 512-byte logical block. Logical block numbers range from 0 to 511.

Requests for physical read/write functions specify the initial tape address in terms of a 128-byte physical record. (That is, tape positioning is performed in special address mode.) Physical record numbers range from 0 to 2047. Multirecord transfers -- exceeding 128 bytes -- read from or write to physically sequential records on the tape.

The functions provided by the handler are described in further detail below.

#### NOTE

DIGITAL provides a special version of the DD handler for the KXT11-C peripheral processor. The KXT11-C DD handler is identical to the DD handler described in this section, except that it includes support for the multiprotocol chip that resides on the KXT11-C. Thus, a TU58 subsystem can be connected to any of the serial lines on the KXT11-C.

The KXT11-C DD handler resides in the KXT11-C driver library, DRVK.OBJ. To use the KXT11-C DD handler, you must edit its prefix file, DDPFXK.MAC, and then use the prefix file to build the KXT11-C DD handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

One DD handler process must exist for each TU58 controller configured on the target system. (Multiple DD processes share the same instruction and pure-data segments but require separate RAM for impure data.) Each handler process has its own request queue semaphore on which it waits for service requests. The request queue names are \$DDA, \$DDB, \$DDC, ...; \$DDA is associated with controller A, \$DDB with controller B, and so forth. If only one TU58 subsystem is configured, the associated queue semaphore is always named \$DDA. One handler process can support two drive units on the same controller, but operations cannot be performed simultaneously on both drives.

The DD handler interfaces with a TU58 subsystem through a serial line interface unit. Read and write operations are effected via byte transfers. Each serial line unit used for a TU58 subsystem must be configured with appropriate device register and interrupt vector addresses (assuming that the factory-configured standard address

## STANDARD DEVICE HANDLERS

assignments are not suitable). The device register and vector addresses used -- whether standard or not -- must be specified in a configuration file that is read and processed during the building of the application software. (Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.)

TU58 tape drives are supplied in single-drive and dual-drive models. The left or only drive is always designated unit 0; the right drive on dual-drive models is always designated unit 1. The unit number specified in the I/O request packet selects the desired drive unit.

### 4.3.1 Functions Provided

The functions provided by the DD handler are listed below, by symbolic and decimal function code:

Code	Function Performed
I\$SRDP (0)	Read physical
I\$RLP (1)	Read logical (equivalent to read physical)
I\$WTP (3)	Write physical
I\$WTI (4)	Write logical (equivalent to write physical)
I\$GET (7)	Get Characteristics

A logical read or write operation transfers data to or from the user's buffer, starting at a 512-byte logical block specified by 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.

A physical read or write operation transfers data to or from the user's buffer, starting at a device address specified by physical record number, which ranges from 0 to 2047. (An analogous scheme is track/sector addressing for an RX02 diskette.) The unit of storage implied by physical I/O operations is the 128-word record; that is, data transfers can start at any physical record 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. Note that the standard record interleaving that is performed for logical I/O is disabled for physical I/O.

The read-with-increased-threshold and write-verify options are requested by means of device-independent function-modifier bit settings; see below.

The Get Characteristics function returns, in the function-dependent portion of the reply message, a block of device-dependent information. The information consists of the codes for device class and type code and the number of logical blocks per unit (512).

## STANDARD DEVICE HANDLERS

### NOTE

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 ISSUNS status code is returned in the function-independent portion of the reply message.

**4.3.1.1 Device-Dependent Function Modifiers** - The DD handler does not recognize any device-dependent function modifiers. The values of modifier bits 6 to 12 of the function word (DP.FUN) are not significant.

**4.3.1.2 Device-Independent Function Modifiers** - If modifier bit FMSBSM of DP.FUN is set, the DD handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3.

If modifier bit FMSINH of DP.FUN is set, the handler does not retry soft data errors; it returns the ISSPAR status code immediately.

If modifier bit FMSDCK of DP.FUN is set to 1 in a read function request, the handler 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 bit FMSDCK of DP.FUN is set to 1 in a write function request, the handler 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 handler, indicating success or failure. That type of write operation can be used to ensure that reliable data can later be recovered.

### 4.3.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a handler request, following field DP.SEM, is described below for each type of function.

## STANDARD DEVICE HANDLERS

**4.3.2.1 Logical Read/Write Functions** - The function-dependent portion of a logical read or logical write request packet (functions 10h, 11h, IF\$RDL or IF\$WTL) is shown below:

DP.DAD -	Logical blk num -----      -----	Portion of the request sent by value
DP.BUF -	Buffer address -----      -----	-----
DP.PAR -	PAP value      -----	Portion of the request sent by reference
DP.LEN -	Buffer length -----      -----	-----

The range of the logical-block-number value is 0 to 511. The unit number may be 0 or 1. The buffer-length value determines the length of the data transfer, in bytes.

**4.3.2.2 Physical Read/Write Functions** - The function-dependent portion of a physical read or physical write request packet (function code IFSRDP or IFSWTP) is shown below:

DP.DAD =	-----+-----    Physical reg num    -----+-----    0    -----+-----	Portion of the request sent by value
DP.BUF =	Buffer address    -----+-----   --   --	Portion of the request sent by reference
DP.PAR =	PAR value    -----+-----   --   --	Portion of the request sent by reference
DP.LEN =	Buffer length    -----+-----   -----+-----	Portion of the request sent by reference

The range of the physical-req-num value is 0 to 2047. The unit number may be 0 or 1. The buffer-length value determines the length of the data transfer, in bytes.

## STANDARD DEVICE HANDLERS

4.3.2.3 Get Characteristics Function - The function-dependent portion of a Get Characteristics request packet is ignored by the handler and may be omitted. In the corresponding reply packet, however, the function-dependent portion of the message contains the following information:

DP.DAD =	Type	Class
	Number of logical blocks	(Least significant)
		(Most significant)

In the information above:

- Class is DCSDSK, for disk device class
- Type is DKSDD, for TU58 device type

The number of logical blocks is given per unit, that is, for one DECTape II cartridge (always 512 blocks).

### 4.3.3 Status Codes

The DD handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Successful completion
ISSDRV	Drive error: all retries failed, data check error, seek error (block not found), motor stopped, or bad op code
ISSFUN	Invalid function code
ISSIBN	Invalid block number: exceeds 511
ISSNXU	Nonexistent unit: drive number greater than 1
ISSUNS	Unsafe volume: no cartridge in drive
ISSWLK	Cartridge is write-protected: write request failure

### 4.3.4 Extended Error Information

For all status returns and especially for an error condition -- DP.STS is nonzero status -- the handler returns additional information in field DP.ERR of the reply message. That information consists of byte 3 of the end packet that is returned to the handler by the TU58 controller for the operation in question. That byte value, in the low-order byte of DP.ERR, is the hardware success code, which provides more specific error information in a hard-error status return.

See the TU58 DECTape II User's Guide (Order No. EK-#TU58-UG-003) for a description of the end packet sent by the tape controller.

## STANDARD DEVICE HANDLERS

### 4.4 RL01/RL02 (DL) DEVICE HANDLER

The DL device handler supports both logical and physical I/O operations on RL01 and RL02 disks. The handler also reports standard device characteristics, including the storage capacity per unit in terms of logical blocks. To perform these functions, the DL handler uses one of three disk controllers. These controllers, and the disks and addressing modes they support, are as follows:

Controller	Disks	Addressing Modes
RLV11	RL01	16- or 18-bit
RLV12	RL01, RL02	16-, 18-, or 22-bit
RLV21	RL01, RL02	16- or 18-bit

Requests to the DL handler for logical read/write functions specify the initial disk address in terms of a logical block number. Multisector logical transfers read from or write to logically sequential sectors of the disk.

Requests for physical read/write functions specify the track, cylinder, and sector at which the transfer is to begin. The device handler converts logical block numbers into physical device addresses -- tracks, cylinders, and sectors. Logical blocks span several sectors and may cross cylinders.

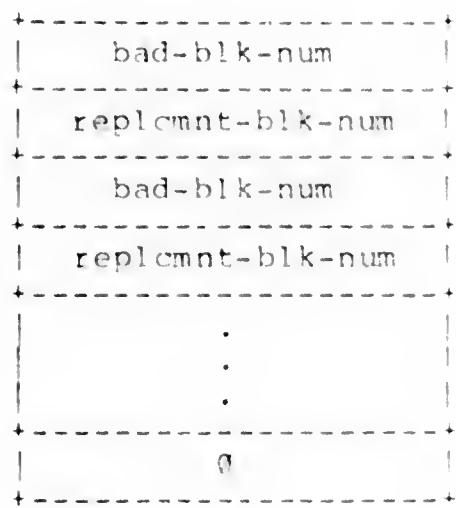
Unlike other standard handlers, the DL handler has a very elementary interrupt service routine (ISR). The ISR signals the interrupt-handling section of the process when an interrupt occurs. Thus, virtually all handler processing for an RL01 or RL02 is performed at process level, not ISR level.

One DL handler process must exist for each RL01/RL02 controller configured on the target system. (Multiple DL processes share the same instruction and pure-data segments but require separate RAM for impure data.) Each handler process has its own request queue semaphore on which it waits for service requests. The request queue names are SDLA, SDLB, SDLC, and so on. SDLA is associated with controller A, SDLB with controller B, and so forth. If only one controller is configured, the associated queue semaphore is named SDLA. One handler process can support four drive units -- RL01s and RL02s in any combination -- on the same controller.

The DL handler interfaces with the RL01 and RL02 hardware via an RLV12, RLV21, or RLV11 controller interface module connected to the target system's backplane. All read and write operations are effected using direct memory access (DMA) transfers. The RLV12 controller supports 16-, 18-, and 22-bit addressing modes; the RLV11 and RLV21 controllers, 16- and 18-bit addressing modes. Each controller module is factory configured with standard device addresses -- one for each addressing mode supported on the device -- and a standard interrupt vector address. When more than one controller is included in a system or if the standard address assignments are not desired, you must reconfigure the controller(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 configuration file that is edited and processed when the application software is built. (See the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide for software configuration procedures.)

## STANDARD DEVICE HANDLERS

The DL handler supports bad-block replacement via 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:



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. Note that 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 DL handler also supports dynamic mounting and dismounting of disk packs. The handler 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.)

### 4.4.1 Functions Provided

The DL handler functions provided are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical
IFSGET (7)	Get Characteristics

A 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. That is, 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.

## STANDARD DEVICE HANDLERS

A logical read or write operation transfers data to or from the user's buffer, starting at a 256-word logical block specified by 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 256-word logical block, which consists of two logically contiguous sectors. 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 handlers reserve these 10 blocks for bad-block replacements. The DL handler 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 the same as that used by the RT-11 RL01 and RL02 handlers: twenty 2-sector logical blocks per track with a 34-sector per track offset. Thus, MicroPower/Pascal disks are RT-11-compatible.

All read and write operations -- physical or logical -- transfer an even number of bytes to or from the user's buffer, due to the word orientation of the device. If an odd-value buffer length is specified in the request (field DP.LEN), the handler returns an ISSIVP (invalid parameter) status code.

The Get Characteristics function returns, in the function-dependent portion of the reply message, a block of device-dependent information. 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. If the disk is not properly mounted for a Get Characteristics request, the handler returns an unsafe volume (ISSUNS) status code.

## STANDARD DEVICE HANDLERS

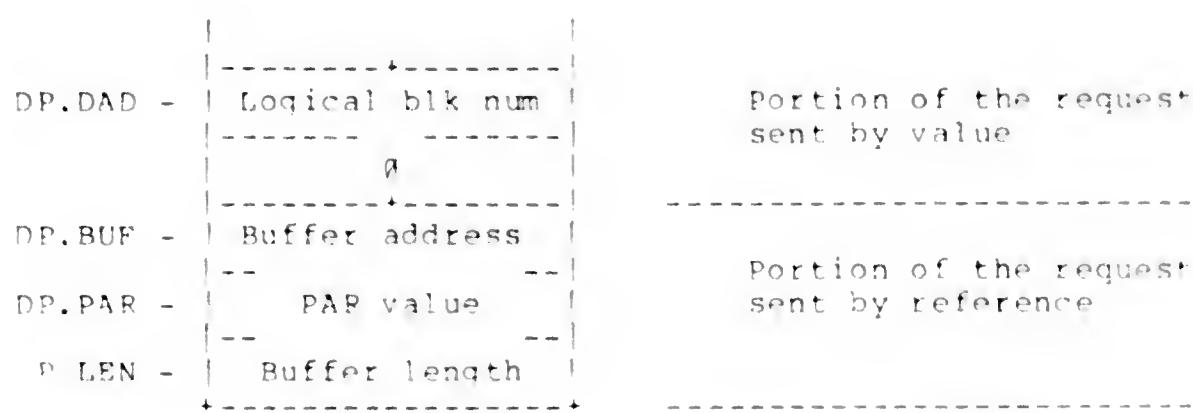
**4.4.1.1 Device-Dependent Function Modifiers** - The DL handler does not recognize any device-dependent function modifiers. The values of modifier bits 6 to 12 of the function word (DP.FUN) are not significant.

**4.4.1.2 Device-Independent Function Modifiers** - If modifier bit FMSBSM of DP.FUN is set, the DL handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3. If modifier bit FMSINH of DP.FUN is set, the handler does not retry soft data errors; it returns an error status code immediately.

#### 4.4.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a DL handler request (following field DP.SEM) is described below for each type of function.

**4.4.2.1 Logical Read/Write Functions** - The function-dependent portion of a logical read or logical write request packet is shown below:



The range of the logical-blk-num value is 0 to 10209 for the RL01 and 0 to 20449 for the RL02. The buffer-length value determines the length, in bytes, of the data transfer.

The desired drive unit is selected by the unit number (DP.UNIT) in the function-independent portion of the request packet. The range of valid unit numbers is specified in the DL driver prefix file; typically, unit numbering starts at 0.

## STANDARD DEVICE HANDLERS

4.4.2.2 Physical Read/Write Functions - The function-dependent portion of a physical read or physical write request packet is shown below:

DP.DAD =	Track   Sector   -----+-----+ Cylinder	Portion of the request sent by value
DP.BUF =	Buffer address	-----+-----+-----+-----+-----+-----+
DP.PAR =	--   PAR value   --   --	Portion of the request sent by reference
DP.LEN =	Buffer length   -----+-----+-----+-----+-----+	

The range of the cylinder-number value is 0 to 255 (RL01) or 0 to 511 (RL02). The track-number value is 0 or 1. The range of the sector-number value is 1 to 40. The buffer-length value determines the length, in bytes, of the data transfer.

The desired drive unit is selected by the unit number (DP.UNIT) in the function-independent portion of the request packet. The range of valid unit numbers is specified in the DL driver prefix file; typically, unit numbering starts at 0.

4.4.2.3 Get Characteristics Function - The function-dependent portion of a Get Characteristics request packet is ignored by the handler and may be omitted. The desired drive unit is selected by the unit number (DP.UNIT) in the function-independent portion of the request packet. The range of valid unit numbers is specified in the DL handler prefix file; typically, unit numbering starts at 0.

In the corresponding reply packet, the function-dependent portion of the message contains the following information:

DP.DAD =	Type   Class   -----+-----+ Number of   Least significant   -----+-----+ logical blocks   Most significant -- always 0   -----+-----+ Tracks   Sectors   -----+-----+ Cylinders   -----+-----+
----------	---

In the information above:

- Class is DCS05E, for disk controller class
- Type is DKSDU, for PCP/PCPD device type

The number of logical blocks, tracks, sectors, and cylinders is 1100 per unit -- for one disk. The number of tracks is reported as 2, indicating two recording surfaces.

## STANDARD DEVICE HANDLERS

### 4.4.3 Status Codes

The DL handler returns the following completion status codes in the DPSTS of the reply message:

ISSNOR	Normal completion
ISSABT	Handler process deleted; request not submitted
ISSDRV	Drive error
ISSFUN	Invalid function code
ISSIDA	Invalid device address on read/write request
ISSIVP	Invalid request packet parameter: odd address for buffer or odd number of bytes to transfer
ISSNXM	Nonexistent memory
ISSNXU	Nonexistent unit; unit number not defined in prefix file
ISSPAR	Parity error, CRC error
ISSUNS	Unsafe volume
ISSWLK	Write-locked unit

### 4.4.4 Extended Error Information

In the event of an error, the DL handler copies the multipurpose register (MPR) into the DPERR field of the reply packet. See the RLV12 Disk Controller User's Guide for a description of the MPR.

## STANDARD DEVICE HANDLERS

### 4.5 MSCP DISK-CLASS (DU) DEVICE HANDLER

The DU device handler supports I/O operations on disks and disk controllers that use the Mass Storage Control Protocol (MSCP) interface. In particular, the handler supports I/O on the RD51 and the RX50, both available on the MICRO/PDP-11. The RD51 is a 10MB fixed Winchester disk, and the RX50 is a dual 400KB 5 1/4-inch diskette drive. The DU handler communicates with the RD51 and the RX50 through the RQDX1 controller, which uses the MSCP interface.

#### NOTE

MSCP is a high-level interface to a family of mass storage controllers and devices manufactured by DIGITAL. Under the MSCP interface, the disk controller concerns itself with the details of I/O handling -- device type, media format and geometry, error recovery -- while the disk handler sends I/O requests and receives responses. The MSCP interface offers the caller a basic subset of I/O requests, described in Chapter 6 of the MSCP Basic Disk Functions Manual (part of the UDA50 Programmer's Documentation Kit).

The DU handler supports logical I/O operations and the reporting of standard device characteristics for MSCP disk-class devices. The handler also allows you to establish direct communication with the port of an MSCP controller, so that you have unlimited access to all the commands in the MSCP basic subset.

The DU handler supports any number of MSCP controllers. One DU handler process must exist for each MSCP controller configured on the target system. (Multiple DU processes share the same instruction and pure-data segments, but require separate RAM for impure data.) Each handler process has its own request queue semaphore on which it waits for service requests. The request queue names are SDUA, SDUB, SDUC, and so on. SDUA is associated with controller A, SDUB with controller B, and so forth. If only one controller is configured, the associated queue semaphore is named SDUA. In the case of the RQDX1 controller, one handler process can support four drive units (RD51=1, RX50=2) on the same controller.

The DU handler interfaces with the RD51 and RX50 hardware via an RQDX1 controller interface module connected to the MICRO/PDP-11 backplane. All read and write operations are effected using direct memory access (DMA) transfers in a 22-bit Q-Bus environment. Each controller module is factory-configured with standard device and interrupt vector addresses. When more than one controller is included in a system or if the standard address assignments are not desired, you must reconfigure the controller(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 configuration file that is edited and processed when the application software is built. (See the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide for software configuration procedures.)

## STANDARD DEVICE HANDLERS

### 4.5.1 Functions Provided

The DU handler functions provided are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDL (1)	Read logical
IFSWTL (4)	Write logical
IFSGET (7)	Get Characteristics
IFSONY (8)	Bypass Only
IFSBYP (9)	Bypass
IFSINT (10)	Initialize Port

A logical read or write operation transfers data to or from the user's buffer, starting at a 256-word 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 256-word 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 RD51 or RX50 transfer an even number of bytes to or from the user's buffer, due to the word orientation of the devices. If an odd-value buffer length is specified in the request (field DP.LEN), the handler returns an invalid parameter (ISSIVP) status code.

The Get Characteristics function returns, in the function-dependent portion of the reply message, a block of device-dependent information. The information consists of the codes for device class and type, the number of logical blocks per unit. Note that the only way to distinguish between the RX50 and the RD51 is by the number of logical blocks per unit. If the disk is not properly mounted for a Get Characteristics request, the handler returns an unsafe volume (ISSUNS) status code.

The Bypass function initializes and gives you direct access to the MSCP port, bypassing the usual MSCP controller services -- unit number translation, packet sequencing, error handling, and so forth. This gives you access to all the features described in the MSCP Basic Disk Functions Manual. To use the Bypass function, the caller sends the handler a pointer to a 52-word area in the caller's process, containing an MSCP response buffer and an MSCP command packet. (The format is shown in Section 4.5.2.4.) A reply packet is placed in the user-provided buffer when the request is completed. Note that it is the responsibility of the calling process to perform packet sequencing, end-packet error handling, and so forth. Also, if any state changes were made, you must ensure the integrity of future I/O requests by forcing a reinitialization of the MSCP port after the direct access.

## STANDARD DEVICE HANDLERS

The Bypass Only function sets direct access mode by causing all commands except Bypass, Bypass Only, and Initialize Port to abort with an illegal function (ISSFUN) status code.

The Initialize Port function reinitializes the MSCP port after a Bypass function. The reinitialization is immediate; any just-issued I/O requests abort with the ISSABT status code. Initialize Port resets the handler to accept all commands, undoing the effect of a Bypass Only request.

**4.5.1.1 Device-Dependent Function Modifiers** - The DU handler does not recognize any device-dependent function modifiers. The values of modifier bits 6 to 12 of the function word (DP.FUN) must be zero.

**4.5.1.2 Device-Independent Function Modifiers** - If modifier bit FMSBSM of DP.FUN is set, the DU handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3. If modifier bit FMSINH of DP.FUN is set, the handler does not retry soft data errors; it returns an error status code immediately.

### 4.5.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a DU handler request (following field DP.SEM) is described below for each type of function.

**4.5.2.1 Logical Read/Write Functions** - The function-dependent portion of a logical read or logical write request packet is shown below:

DP.DAD -	Logical blk num	Portion of the request sent by value
	0	
DP.BUF -	Buffer address	
DP.PAR -	PAR value	Portion of the request sent by reference
DP.LEN -	Buffer length	

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.

The desired drive unit is selected by the unit number (DP.UNI) in the function-independent portion of the request packet. The range of valid unit numbers is specified in the DU driver prefix file; typically, unit numbering starts at 0.

## STANDARD DEVICE HANDLERS

**4.5.2.2 Get Characteristics Function** - The function-dependent portion of a Get Characteristics request packet is ignored by the handler and may be omitted. The desired drive unit is selected by the unit number (DP.UNI) in the function-independent portion of the request packet. The range of valid unit numbers is specified in the DU handler prefix file; typically, unit numbering starts at 0.

In the corresponding reply packet, the function-dependent portion of the message contains the following information:

DP.DAD -	-----+-----	
	-----+-----	Type   Class
	-----+-----	Number of   (Least significant)
	-----+-----	logical blocks   (Most significant -- always 0)

In the information above:

- Class is DCSDSK, for disk device class
- Type is DKSDU, for MSCP disk-class device type

The number of logical blocks is given per unit -- for one disk.

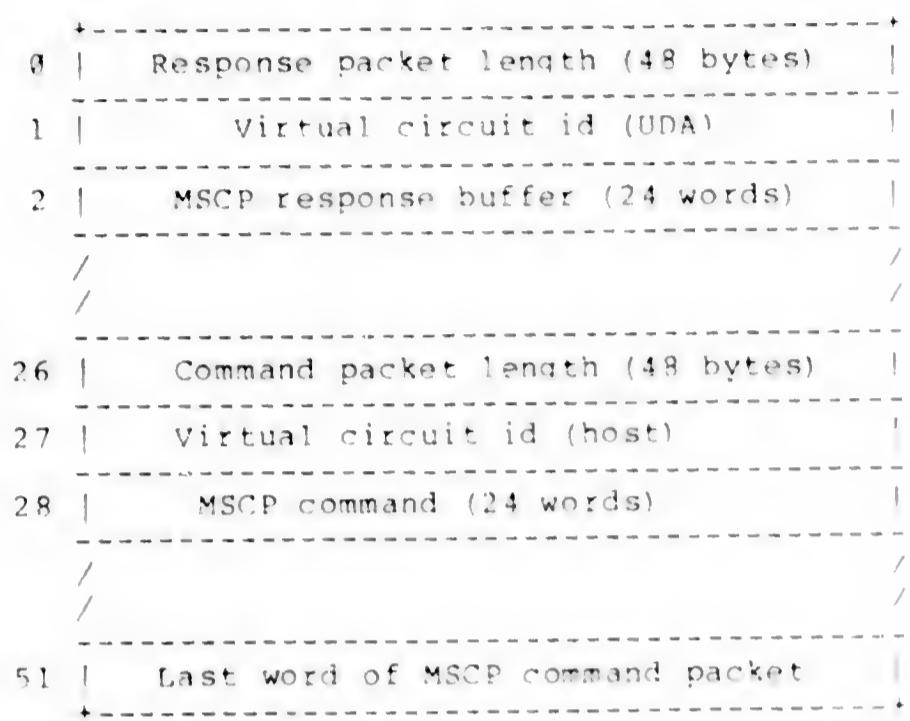
**4.5.2.3 Bypass Only Function** - The function-dependent portion of a Bypass Only request packet is ignored by the handler and may be omitted.

**4.5.2.4 Bypass Function** - The function-dependent portion of a Bypass request packet is shown below:

DP.DAD -	-----+-----	
	-----+-----	-- (Unused) --  Portion of the request sent by value
	-----+-----	
DP.BUF -	-----+-----	-----
DP.PAR -	-----+-----	-----  Portion of the request sent by reference
DP.LEN -	-----+-----	-----

## STANDARD DEVICE HANDLERS

The buffer address specifies a 52-word area in the caller's process. The format of this area is as follows:



The 24-word MSCP command area contains an MSCP request packet; request-packet formats for the MSCP disk commands are described in detail in the MSCP Basic Disk Functions Manual. The 24-word response buffer area receives the MSCP reply packet.

**4.5.2.5 Initialize Port Function** - The function-dependent portion of an Initialize Port request packet is ignored by the handler and may be omitted.

### 4.5.3 Status Codes

The DU handler returns the following completion-status codes in field DP.STS of the reply message:

ISSNOR	Normal completion
ISSABT	Handler process deleted; request not serviced
ISSCTL	Controller error
ISSDRV	Drive error
ISSFOR	Media format error
ISSFUN	Invalid function code
ISSIBN	Invalid block number on read/write request
ISSIVD	Invalid data
ISSIVP	Invalid request packet parameter: odd address for buffer or odd number of bytes to transfer

**STANDARD DEVICE HANDLERS**

ISSNXU	Nonexistent unit; unit number not defined in prefix file
ISSOFL	Unit off line
ISSUNS	Drive not ready, door open, power not on, no volume
ISSWLK	Write-locked unit

## STANDARD DEVICE HANDLERS

### 4.6 RX02 (DY) DEVICE HANDLER

The DY device handler uses an RXV21 controller to support both logical and physical I/O operations on single- or double-density floppy diskettes. The handler also initializes (formats) a diskette for single- or double-density operation and reports standard device characteristics, including the storage capacity per unit in terms of logical blocks.

Requests for logical read/write functions specify the initial device address in terms of a logical block number. Multiple DMA transfers read from or write to logically sequential, but physically discontinuous, sectors of the diskette.

Requests for physical read/write functions specify the cylinder and sector at which the transfer is to begin. The device handler converts logical block numbers into physical device addresses -- cylinders and sectors. Logical blocks span several sectors and may cross cylinders.

The functions provided by the handler are described in further detail below.

One DY handler process must exist for each RX02 controller configured on the target system. (Multiple DY processes share the same instruction and pure-data segments but require separate RAM for input data.) Each handler process has its own request queue semaphore on which it waits for service requests. The request queue names are SDYA, SDYB, SDYC, ...; SDYA is associated with controller A, SDYB with controller B, and so forth. If only one controller is configured, the associated queue semaphore is always named SDYA. One handler process can support two drive units on the same controller.

The DY handler interfaces with RX02 hardware via an RXV21 controller interface module in the target system's backplane. All read and write operations are effected using DMA transfers. Each RXV21 controller module is factory-configured with a standard device address and a standard interrupt vector address. When more than one RXV21 controller is included in a system or if the standard address assignments are not desired, you must reconfigure the controller(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 both in the DY handler prefix file (DYPFX.MAC) and in a configuration file that is edited and processed during the building of the application software. (See the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide for software configuration procedures.)

RX02 floppy disk drives are supplied in single-drive and dual-drive models. The left (or only) drive is always designated unit 0; the right drive on dual-drive models is always designated unit 1. The unit number specified in the I/O request packet selects the desired drive unit.

## STANDARD DEVICE HANDLERS

### 4.6.1 Functions Provided

The functions provided by the DY handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical
IFSGET (7)	Get Characteristics

A 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. That is, 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 a diskette for single-density or double-density operation. (See Section 4.6.1.1.)

A logical read or write operation transfers data to or from the user's buffer, starting at a 256-word logical block specified by a logical block number -- 0 to 193 for single density, 0 to 987 for double density. The unit of storage implied by logical I/O operations is the 256-word 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.) 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 a 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 the same as that used by RT-11 RX02 handlers: 2:1 interleaving with a 6-sector per cylinder offset. Thus, MicroPower/Pascal diskettes are RT-11-compatible.)

All read or write operations, physical or logical, transfer an even number of bytes to or from the user's buffer, due to the word orientation of the device. Therefore, if an odd-value buffer length is specified in the request (field DP.LEN), the handler assumes length-1 bytes as the effective transfer length.

## STANDARD DEVICE HANDLERS

All user-requested read or write operations, physical or logical, 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 ISSIVM 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 ISSIVM status code is returned to the application program.

The Get Characteristics function returns, in the function-dependent portion of the reply message, a block of device-dependent information. 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. If the diskette is not properly mounted for a Get Characteristics request, the handler returns an unsafe volume (ISSUNS) status code.

**4.6.1.1 Device-Dependent Function Modifiers** - If modifier bits FMSWFM and FMSWSD of the function word are both set in a write physical (IFSWTP) function request, the meaning of the function is "format diskette for single density." If modifier bit FMSWFM is set and modifier bit FMSWSD is not set in a write physical request, the meaning of the function is "format diskette for double density." The device-dependent function modifiers (bits 6 to 12 of word DP.FUN) have no significance for any other functions.

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.

**4.6.1.2 Device-Independent Function Modifiers** - If modifier bit FMSBSM of DP.FUN is set, the DY handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3. If modifier bit FMSINH of DP.FUN is set, the handler does not retry soft data errors; instead, it returns the ISSPAR status code immediately.

## 4.6.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a DY handler request (following field DP.SEM) is described below for each type of function.

## STANDARD DEVICE HANDLERS

4.6.2.1 **Logical Read/Write Functions** - The function-dependent portion of a logical read or logical write request packet is shown below:

DP.DAD -	-----+-----    Logical blk num    -----+-----    0    -----+-----
DP.BUF -	-----+-----    Buffer address    -----+-----    --    -----+-----
DP.PAR -	-----+-----    PAR value    -----+-----    --    -----+-----
DP.LEN -	-----+-----    Buffer length    -----+-----

The range of the logical-blk-num value is 0 to 493 for single density and 0 to 987 for double density. The buffer-length value determines the length of the data transfer, in bytes.

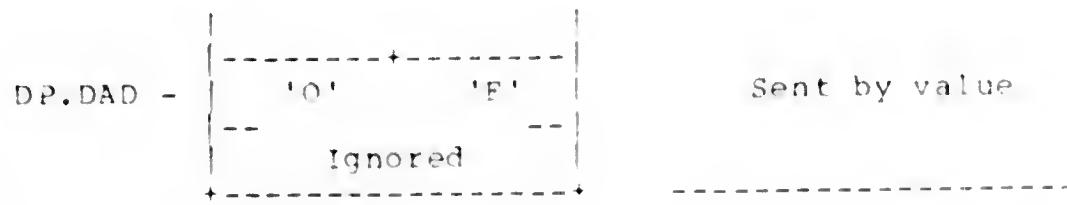
4.6.2.2 **Physical Read/Write Functions** - The function-dependent portion of a physical read or physical write request packet is shown below:

DP.DAD -	-----+-----    Track   Sector    -----+-----    Cylinder    -----+-----
DP.BUF -	-----+-----    Buffer address    -----+-----    --    -----+-----
DP.PAR -	-----+-----    PAR value    -----+-----    --    -----+-----
DP.LEN -	-----+-----    Buffer length    -----+-----

The range of the cylinder-number value is 0 to 76. The track number value must be 0. The range of the sector number value is 1 to 26. The buffer-length value determines the length of the data transfer, in bytes.

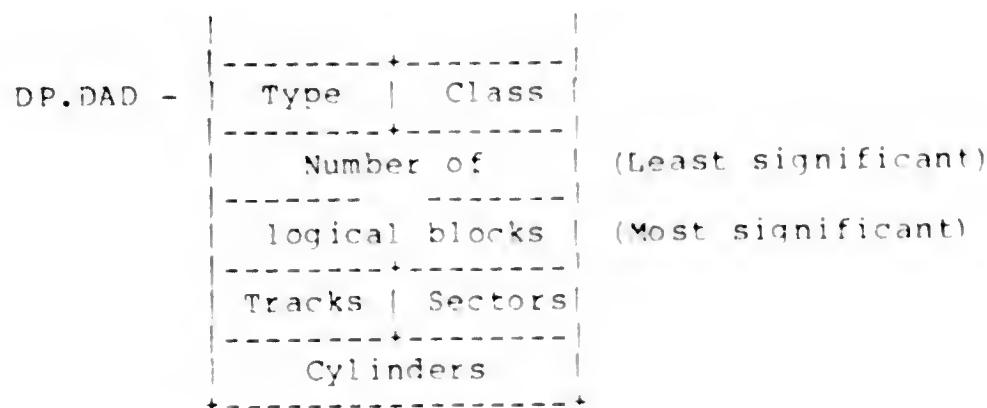
## STANDARD DEVICE HANDLERS

4.6.2.3 Format Subfunctions of Physical Write - The function-dependent portion of a request packet for the single- and double-density formatting subfunctions of physical write is shown below:



The DP.DAD field must contain the ASCII character sequence F0 in the first (low-order) word.

4.6.2.4 Get Characteristics Function - The function-dependent portion of a Get Characteristics request packet is ignored by the handler and may be omitted. In the corresponding reply packet, however, the function-dependent portion of the message contains the following information:



In the information above:

- Class is DCSDSK, for disk device class
- Type is DKSDY2, for RX82 device type

The number of logical blocks, tracks, sectors, and cylinders is given per unit, that is, for one diskette. (The number of tracks is reported as 1, indicating a single recording surface.)

## STANDARD DEVICE HANDLERS

### 4.6.3 Status Codes

The DV handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal completion
ISSABT	Handler process deleted; request not serviced
IS\$PUN	Invalid function code
ISSIVM	Invalid mode: volume formatted for opposite or unrecognized density
ISSIVP	Invalid request packet parameter: odd number of bytes to transfer
ISSNXM	Attempted transfer to nonexistent memory or attempted write to read-only memory
ISSNXU	Nonexistent unit: drive number greater than 1
ISSPAR	Unrecoverable CRC error or soft error with no retry
ISSPWR	Power failure
ISSUNS	Drive not ready: door open, power not OK, drive not up to speed

### 4.6.4 Extended Error Information

In the event of a hardware error, extended error information is returned in the DP.ERR field of the reply packet. This information consists of one byte of definitive error code as returned by the RX211/RXV21 in response to the read error code function. This status information is described in the RX02 Floppy Disk System User's Guide.

## STANDARD DEVICE HANDLERS

### 4.7 KWVII-C (KW) REAL-TIME CLOCK HANDLER

The KW handler supports I/O operations on the KWVII-C real-time clock board. The KWVII-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. 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 KWVII-C clock has two Schmitt triggers. They can, in response to external events, start the clock, start analog-to-digital (A/D) conversions in an ADVII-C or AXVII-C A/D converter (see the AA handler section), or generate program interrupts to the processor.

The KW handler consists of a request-handling static process and several routines, including signaling and external event interrupt service routines (ISRs). The KW handler does not support multiple controllers; only one KW handler static process can be resident in a target system. The KW handler process has a request queue semaphore on which it waits for I/O requests. The handler's request queue semaphore name is SKWA. The handler process also maintains an internal read-request queue used to handle overlapping requests.

Each KWVII-C board is factory-configured with a standard device address and a standard interrupt vector address. When the standard address assignments are not desired, you must reconfigure the PW controller 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 configuration file that is edited and processed during the building of the application software. (See the MicroPower Pascal-RT System User's Guide and the MicroPower Pascal-PSV/VMS System User's Guide for software configuration procedures.)

#### NOTE

As an alternative to the standard KW handler interface described in this section, DIGITAL supplies four MicroPower/Pascal procedures that support the KWVII-C real-time clock: Read counts wait, Read counts signal, Start clock, and Stop clock. See Appendix G for detailed descriptions of these procedures.

#### 4.7.1 Functions Provided

The KW handler functions provided are listed below, by name and decimal function code:

## STANDARD DEVICE HANDLERS

Code	Function Performed
IFSRDP (0)	Read physical
IFSGET (7)	Get Characteristics
IFSSTC (8)	Start real-time clock
IFSSTR (9)	Stop real-time clock

The read physical function reads the clock's buffer preset register after each clock interrupt until the buffer is filled or an error occurs.

The Get Characteristics function returns the object's physical characteristics.

The stop real-time clock function stops the KWV11-7 Programmable Real-Time Clock by disabling interrupts on the device.

The start real-time clock function initializes the KWV11-7 Programmable Real-Time Clock.

**4.7.1.1 Device-Dependent Function Modifiers** - The KW handler does not use device-dependent function modifier bits of the function word (DP.FUN).

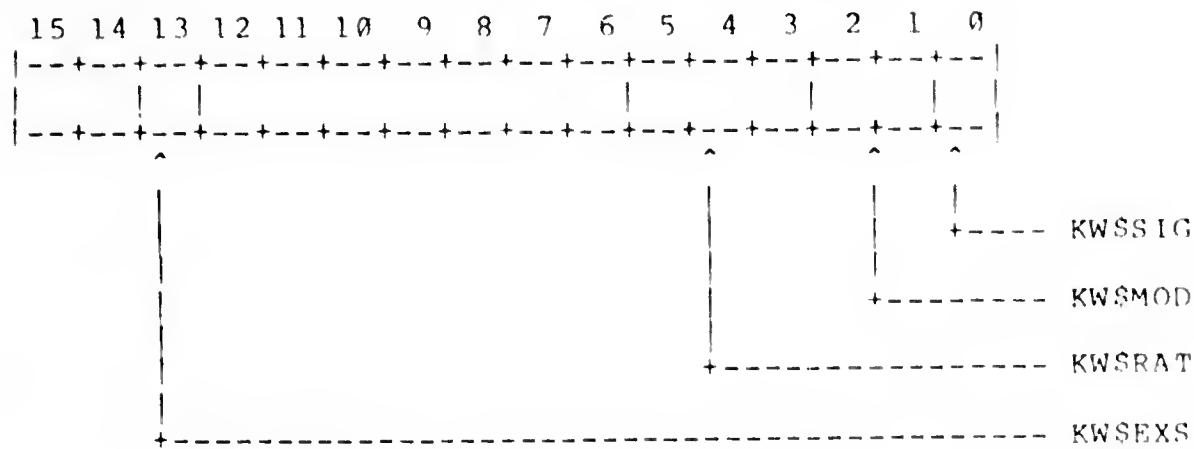
**4.7.1.2 Device-Independent Function Modifiers** - If modifier bit FMSBSM of DP.FUN is set, the KW handler signals a binary semaphore or a counting semaphore, as described in Sections 4.1.2 and 4.1.3.

### 4.7.2 Function-Dependent Request Formats

The function-dependent portion of a KW handler request (following field DP.SEM) is described below for each type of function. Both the read physical request and the start real-time clock request require the clock control word (DP.KWF) and the clock count value (DP.KWC). These two parameters are described below.

The clock control word (DP.KWF) is in the location where the low-order portion of a device address is normally found. The format of the clock control word is as follows:

## STANDARD DEVICE HANDLERS



The bit fields of the clock control word (DP.KWF) control the actions of the clock.

Field	Usage
KW\$SIG	This 1-bit field is set to 1 when a binary or a counting semaphore is to be signaled after each clock interrupt. When the clock is being used to trigger A/D conversions, this field is set to 0. This field is valid only with the start real-time clock request.
KW\$MOD	This 2-bit field specifies the clock's mode of operation. For the start real-time clock request, the two correct values for this field are 0 and 1. For the read physical request, the two correct values for this field are 2 and 3.
<b>Mode      Operation</b>	
0	Single interval
1	Repeated interval
2	External event timing
3	External event timing from zero base

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.

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 ST2 (Schmitt trigger 2) causes the contents of the counter to be loaded into the BPR (buffer/preset register), where it can be read by the KW device handler.

## STANDARD DEVICE HANDLERS

Mode 3 (external-event timing from zero base) is the same as mode 2 (external-event timing) except that the clock is reset to 0 after each event.

### KWSRAT

The 3-bit field KWSRAT selects 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 line (50/60 Hz)
7	

### KWSEXS

This 1-bit field, when set to 1, specifies that the clock is to be started by an external event (Schmitt trigger 2). If this field is set to 0, the KW device handler will start the clock immediately.

The clock count value (DP.KWC) is in the high-order portion of a device address. The KW device handler copies the two's complement of this value to the clock's buffer/preset register. You supply the KW driver with one of the following counts:

- 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

**4.7.2.1 Read Physical Function** - This request, used when the clock is monitoring external events, can either record the time of external events or count external events. The function-dependent portion of read physical request (IF\$RD\$P) is shown below:

DP.KWF -	Clock control	Portion of the request sent by value
DP.KWC -	Count	
DP.BUF -	Buffer	
DP.PAR -	PAR value	Portion of the request sent by reference
DP.LEN -	Buffer size	

The buffer address (DP.BUF), which must be on a word boundary, specifies the location of the data buffer.

## STANDARD DEVICE HANDLERS

4.7.2.2 Get Characteristics Function - The Get Characteristics request (IF\$GET) returns the following information in the function-dependent portion of the reply packet:

DP.FDD -	-----+-----
	Type   Class
	-----+-----

In the information above:

- Class is DCSRLT, for real-time device class
- Type is RTSKWV, for the KWV11-C

4.7.2.3 Start Real-Time Clock Function - The start real-time clock request (IFSSPC), used to initialize the KWV11-C, starts the clock for either a single interval or a repeated interval. This function can signal either a binary or a counting semaphore after each interval or can trigger an A/D conversion after each interval. The function-dependent portion of a start real-time clock request (IFSSTC) is shown below:

DP.KWF -	-----+-----	Portion of the request sent by value
	-----+-----	
DP.KWC -	-----+-----	-----
	-----+-----	
	-----+-----	Portion of the request sent by reference
DP.KWS -	-----+-----	
	-----+-----	
	-----+-----	

4.7.2.4 Stop Real-Time Clock Function - The stop real-time clock request (IFSSTC) disables interrupts on the KWV11-C. This request is used only in conjunction with the start real-time clock request (IFSSPC). (See Section 4.7.2.3.) The stop real-time clock request has no function-dependent portion; the necessary information is contained in the function word (DP.FUN).

## STANDARD DEVICE HANDLERS

### 4.7.3 Status Codes

The KW handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal completion
ISSABT	Handler process deleted; request not serviced
ISSFUN	Invalid function code
ISSOVF	Overflow occurred
ISSOVR	Overrun occurred

### 4.7.4 Extended Error Information

In the event of a hardware error, the contents of the hardware command status register (CSR) are copied into the reply packet, beginning at word DP.FDD. This information consists of two bytes of status. See the LSI-11 Analog System User's Guide for details on the contents of the hardware CSR.

## STANDARD DEVICE HANDLERS

### 4.8 KXT11-C TWO-PORT RAM (KX) HANDLER

The KXT11-C is a peripheral processor for LSI-11-bus-based systems. It functions in a master/slave relationship with an arbiter processor running on the LSI-11 bus. The arbiter (master) controls all interactions between the two processors. The KXT11-C (slave) waits for a command from the arbiter processor before initiating any message transfer. The mechanism of arbiter/KXT11-C interactions is a simple request-reply protocol operating through the dual-port registers of the KXT11-C. That protocol is supported by the KX handler, which runs on the arbiter, and the KK handler (see the KK handler section), which runs on the KXT11-C.

#### NOTE

As an alternative to interfacing directly to the KX handler as described in this section, DIGITAL supplies two MicroPower/Pascal functions that support the arbiter side of the arbiter/KXT11-C protocol: `KX_write_data` and `KX_read_data`. See Appendix B for detailed descriptions of those functions.

An arbiter/KXT11-C transaction occurs as follows: First, the arbiter interrupts the KXT11-C by issuing a command. By issuing the command, the arbiter causes ownership of the dual-port registers to switch to the KXT11-C. The KXT11-C then reads the command, checks its validity, and acts on the command. After acting on the command, the KXT11-C indicates -- by relinquishing control of the dual-port registers and, if interrupts are enabled, by issuing an interrupt to the arbiter -- that the command has been processed.

The KX and KK handlers implement the protocol via read and write commands that they issue to each other. Each KX read or write operation transfers data between an arbiter buffer and a KXT11-C process.

#### NOTE

See Appendix A of the KXT11-C Peripheral Processor Software User's Guide for a detailed description of the KX/KP communication protocol.

The KX handler supports up to 14 KXT11-Cs running on the LSI-11 bus. The handler communicates with the KXT11-C via the command and status registers of Data Channel 0 (4-byte data area) and Data Channel 1 (12-byte data area) of the KXT11-C dual-port RAM. Each data channel is configured in your application -- one or two per KXT11-C -- is assigned a unit number by the KX handler for purposes of communication. Each unit is associated with a unique interrupt to the KX handler to permit fast communication.

Table 4-4 shows the TPP data channel addresses associated with each KXT11-C ID. Note that for each KXT11-C in your system, you must select an ID number that is unique in the system and either a high or low base address range.

## STANDARD DEVICE HANDLERS

**Table 4-4**  
**TWO-PORT RAM Data Channel Addresses**

KXT11-C ID Switch Position	High-Range Channel <sup>1</sup>		Low-Range Channel <sup>1</sup>	
	Address (Jumper In)	Address (Jumper Out)	Address (Jumper In)	Address (Jumper Out)
Channel 0	Channel 1	Channel 0	Channel 1	
0	S T A N D - A L O N E	M O D E	S T A N D - A L O N E	M O D E
1	S T A N D - A L O N E	M O D E	S T A N D - A L O N E	M O D E
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 4-5 shows the default controller ID, CSR address, and interrupt vector values that are supplied in the KX handler prefix file (as distributed on the MicroPower Pascal kit). The prefix file assumes a high base address range for KXT11-C's 2 through 7 and a low base address range for KXT11-CS's 8 through 15.

**Table 4-5**  
**KX Prefix File Details**

KXT11-C ID Switch Position	Default Controller		Default Unit 1	
	CSR Address	CSR & Vector	CSR Address	CSR & Vector
CSRXX	CSR	Vector	CSR	Vector
0	S T A N D - A L O N E	M O D E	S T A N D - A L O N E	M O D E
1	S T A N D - A L O N E	M O D E	S T A N D - A L O N E	M O D E
2	162110	500	162120	514
3	162150	510	162160	514
4	162210	520	162220	524
5	162250	520	162260	524
6	162310	530	162320	534
7	162350	530	162360	534
8	175410	560	175420	564
9	175450	570	175460	574
10	175510	600	175520	614
11	175550	610	175560	614
12	175610	620	175620	624
13	175650	620	175660	624
14	175710	640	175720	644
15	175750	650	175760	654

## STANDARD DEVICE HANDLERS

Each KXT11-C configured adds approximately 581NSR bytes to the handler's input data area. Using the set of code provided adds 46(512) bytes to the handler's input data area.

To use the RX handler, you must edit its prefix file, IFSRX.H, and then use the prefix file to build the RX handler into your application software. Software configuration procedures are described in the MicroPower Pascal-RX System User's Guide and the MicroPower Pascal-RX VM System User's Guide.

### 4.8.1 Functions Provided

The functions provided by the RX handler include the following function codes:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical (current) device physical
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical (current) device physical
IFSGFT (7)	Get characteristics

The read functions IFSRDP and IFSRDL instruct the RX handler to read data from a specified unit and place it in a user-specified data area. If necessary, the request is held until the KXT11-C with the specified unit number becomes ready.

The write functions IFSWTP and IFSTWL instruct the RX Handler to send data to a specified unit from a user-specified data area. If necessary, the request is held until the KXT11-C with the specified unit number becomes ready.

The Get Characteristics function returns, in the function-independent portion of the reply message, bit settings that indicate the device class and the device type of the RX handler.

#### 4.8.1.1 Device-Independent Function Modifiers - IF modifier FMSINH (bit 13) of DP.FUN is set, the RX handler signals a priority interrupt semaphore, as described in Sections 4.1.1 and 4.1.2.

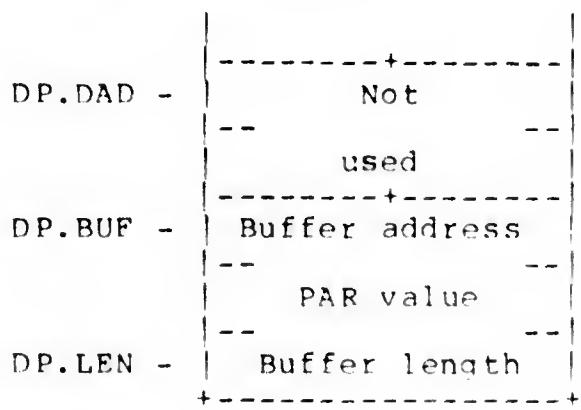
If modifier FMSINH (bit 13) is set, replies of read and write requests are inhibited. If the unit specified cannot complete the request immediately, the RX handler returns an error in the reply packet without retrying the request.

### 4.8.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a RX handler request, following field DP.CFM, is described below for each type of function.

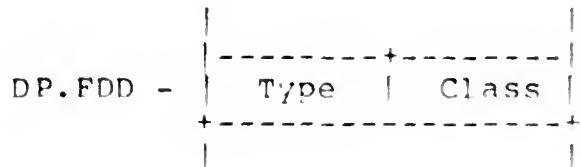
## STANDARD DEVICE HANDLERS

4.8.2.1 **Read or Write Functions** - The function-dependent portion of a read or write request -- function code IFSRDP, IFSRDL, IFSWTP, or IFSWTL -- is shown below:



The desired unit is selected by the unit number (DP.UNI) in the function-independent portion of the request packet. See the unit number table in Section 4.8.4.

4.8.2.2 **Get Characteristics Function** - The Get Characteristics request returns the following information in the function-dependent portion of the reply packet:



In the information above:

- Class is DC\$PRL, for protocol device class
- Type is PRSKKX, for arbiter two-port-RAM handler

### 4.8.3 Status Codes

The KX handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal completion
ISSABT	Handler aborted; no further request can be made to the handler
ISS\$FUN	Invalid function code
ISSNXU	Invalid unit specification
ISS\$OFL	Device off line: KXT11-C was not ready
ISSOVF	Data not available: KXT11-C rejected read
ISSOVR	No data requested: KXT11-C rejected write

## STANDARD DEVICE HANDLERS

### 4.9 DRV11-J PARALLEL-LINE (XA) DEVICE HANDLER

The XA device handler supports I/O operations on devices connected to four 16-bit ports (A, B, C, and D) of a DRV11-J (high-density) parallel-line interface unit. Ports B, C, and D provide identical 16-bit parallel I/O interfaces to user devices.

In the standard, factory-jumpered DRV11-J configuration, port A provides 12 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 handler prefix file (XAPFX.MAC) 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 handler 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 JSRPRI from 1 to 0, as indicated in the source comments, and reassemble the prefix file before installing it in the system.

The handler performs block-mode input and output operations. A specified (even) number of bytes are transferred, by word, to or from the requestor's data buffer per I/O request. The unit numbers for read and write functions are 1, 2, and 3, corresponding to ports B, C, and D, respectively. (Unit number 0 corresponds to port A, if used as an I/O port.)

The enable function unmasks a given interrupt-signal line of port A and logically attaches it to a user-specified semaphore; thus, the semaphore is signaled each time the corresponding interrupt occurs (line goes active-low). The disable function masks off a given previously enabled interrupt-signal line. The unit numbers for the enable and disable functions are 4 to 15, corresponding to lines 0 to 11, respectively, of port A.

Each DRV11-J interface unit is factory configured with standard device register addresses. The handler sets the required interrupt vector

## STANDARD DEVICE HANDLERS

addresses during system power-on. (The DRV11-J vector addresses are not jumper configurable and must be established under program control.) When more than one DRV11-J 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 addresses. In any case, the device register addresses used and the vector addresses desired -- whether standard or not -- must be specified in a configuration file that is edited and processed during the building of the application software; the vector addresses must be contiguous. (Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and in the MicroPower/Pascal-RSX/VMS System User's Guide.)

### 4.9.1 Functions Provided

The functions provided by the XA handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRLP (1)	Read logical (equivalent to read physical)
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical (equivalent to write physical)
IFSGET (7)	Get Characteristics
IF\$ENA (8)	Enable
IFSDSA (9)	Disable

For parallel-line service, there is no distinction between physical and logical function requests; either function code may be used.

The read and write functions transfer only an even number of bytes to or from the user's buffer; data is transferred by word. Therefore, if an odd buffer length is specified in the request (field DP.LEN), the handler assumes length-1 bytes as the effective transfer length.

The enable function enables interrupts on a specified interrupt line (unit 4 to 15), if the line is masked, and associates the user's signal semaphore with that line. (The signal semaphore is distinct from the user's reply semaphore.) The signal semaphore may be a binary/counting semaphore or a queue semaphore, as indicated by function-modifier bit 6; see below. If the specified interrupt line is already enabled when the request is issued, the enable function returns a "line in use" error status. (A disable request must intervene between two enable requests for the same line.)

The 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 particular effect and returns a normal status.

**4.9.1.1 Device-Dependent Function Modifiers** - The XA handler does not recognize any device-dependent function modifiers for the read, write, or disable functions. The values of modifier bits 6 to 12 of the function word (DP.FUN) are not significant for these functions.

## STANDARD DEVICE HANDLERS

4.9.1.2 Device-Independent Function Modifiers - If bit 13 of DP.FUN is set, the XA handler sends a full reply to the requestor, as described in Sections 4.1.2 and 4.1.3. The setting of bit 15 of DP.FUN (inhibit soft-error retry) is not meaningful for parallel-line service and is ignored by the XA handler.

### 4.9.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of an XA handler request (following field DP.SEM) is described below for each type of function.

#### 4.9.2.1 Read or Write Functions - The function-dependent portion of a read or write request is shown below:

DP.SGL -	-----+-----    (Unused)    -----+-----    (Unused)    -----+-----    (Unused)    -----+-----	Portion of the request sent by value
DP.BUF -	Buffer address    -----+-----    --     --	-----
DP.PAR -	FAR value    -----+-----    --     --	Portion of the request sent by reference
DP.LEN -	Buffer length    -----+-----	-----

The unit number selects the desired I/O port; the range of valid unit numbers is 0 to 3 for ports A to D, respectively. 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 of the data transfer, in bytes. If the length value is an odd number, the last byte is not transferred.

#### 4.9.2.2 Get Characteristics Function - In the reply packet of the Get Characteristics request, the function-dependent portion of the message contains the following information:

DP.FDD -	-----+-----    Type   Class  -----+-----   -----+-----    Subtype    -----+-----
----------	---

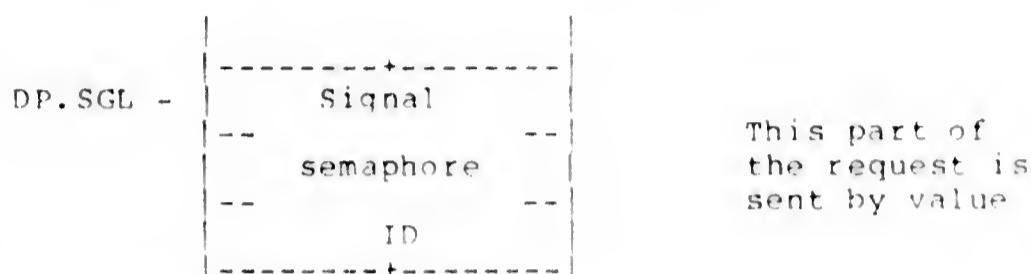
In the information above:

- Class is DCSRLT, for real-time device class

## STANDARD DEVICE HANDLERS

- Type is RT\$DRJ, for DRV11-J device type
- Subtype is dependent on the unit number:
  - RSSSLN = sense-line subtype
  - RSSPRT = parallel-port subtype

4.9.2.3 **Enable Function** - The function-dependent portion of an enable request (function code IFSENA) is 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 14 for port A lines 0 to 11, respectively.

4.9.2.4 **Disable Function** - The entire function-dependent portion of a disable function request (function code IFSDSA) is ignored by the handler and may be omitted for this function.

### 4.9.3 Status Codes

The XA handler returns the following completion-status codes in either field DP.STS of a reply message or the attention code of a signal message.

Code	Meaning
ISSNOR	Successful completion; block transmission completed, or attach/detach operation performed
ISSART	Handler process deleted; request not serviced
ISSDAL	Device allocated, or line already attached (in reply to an attach request)
ISSFUN	Invalid function code or invalid unit number for requested function
ISSNXU	Nonexistent unit number: not in range of 0 to 15.

For ISSNOR status returns, field DP.ALN of the reply message contains the number of bytes transferred.

No information is returned in field DP.ERP of the reply message.

## STANDARD DEVICE HANDLERS

### 4.10 SERIAL LINE (XL) DEVICE HANDLER

The XL device handler 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 SXV11-B serial lines, and the SRC-11/21 serial lines are supported by the XL handler.

#### NOTE

DIGITAL provides a special version of the XL handler for the KXT11-C peripheral processor. The KXT11-C XL handler, described in Section 4.17, is identical to the XL handler described in this section, except that it includes support for the multiprotocol chip that resides on the KXT11-C. The KXT11-C XL handler can concurrently service up to three serial I/O ports on the KXT11-C.

The KXT11-C XL handler resides in the KXT11-C driver library, DRVK.OBJ. To use the KXT11-C XL handler, you must edit its prefix file, XLPFXK.MAC, and then use the prefix file to build the KXT11-C XL handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and in the MicroPower/Pascal-RSX/VMS System User's Guide.

The XL handler 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 requestor's buffer space, per read request. In block-mode output, a specified number of bytes are transferred from the requestor'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 handler continuously transmits bytes from the serial line unit to an input ring buffer specified by the requestor. 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 handler continuously transmits bytes from an output ring buffer, as they become available, to the specified serial line unit. The requestor 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 handler provides optional X-OFF/X-ON control character processing. If requested, the handler inhibits the output side of a given serial line unit when it detects an X-OFF character on

## STANDARD DEVICE HANDLERS

the input side and reenables the output when a subsequent byte character is received.

For interface units that support modern control (DLVII-P), the handler allows you to enable data-set interrupts and to receive data-set status information, analogous to RCSR contents, when such interruptions occur. The handler 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 handler 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 handler's request queue semaphore name is SYLA. The logical serial line is specified in the function request by a configuration-determined unit number.

Each DLVII 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 configuration file that is edited and processed during the building of the application system. Software configuration procedures are described in the MicroPower Pascal-RT System User's Guide and in the MicroPower Pascal-RSX VMS System User's Guide.

### 4.10.1 Functions Provided

The functions provided by the XL handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical
IFSSET (6)	Set Status
IFSGFT (7)	Get Status
IFSCRR (8)	Connect Receive Ring Buffer
IFSCXR (9)	Connect Transmit Ring Buffer
IFSDRPP (10)	Disconnect Receive Ring Buffer
IFSDXRP (11)	Disconnect Transmit Ring Buffer
IFSRSC (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.

## STANDARD DEVICE HANDLERS

**4.10.1.1 Read Function** - The IFSRDP or IFSRDL function is performed in block mode. The handler 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.

**4.10.1.2 Write Function** - The IFSWTP or IFSWTL 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
FMSXCK (11)	Enable automatic X-OFF/X-ON processing if set; disable same if cleared

For automatic X-OFF/X-ON processing, the handler intercepts any X-ON and X-OFF characters that occur in the input from a given interface unit. Receipt of an X-OFF character causes the handler to inhibit output from the same interface unit until a subsequent X-ON character is received. The output line is initially assumed to be enabled. (Multiple successive X-ON or X-OFF characters have the same effect as one such character.) If X-OFF/X-ON processing is not requested, the handler passes all input characters to the user.

Write requests to a unit that has been connected to a ring buffer are not supported.

**4.10.1.3 Connect Receive Ring Buffer Function** - The IFSCRR 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.

## STANDARD DEVICE HANDLERS

**4.10.1.4 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.

**4.10.1.5 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
FMSXCK (11)	Enable automatic X-OFF/X-ON processing if set; disable same if cleared

X-OFF/X-ON processing is performed as described above for block-mode operations.

**4.10.1.6 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 Transmit Buffer request.

**4.10.1.7 Report Data-Set Status Change Function** - The IFSRSC function allows the requestor to wait for a change of data-set (modem) status to occur on a specified serial line unit. When such a change occurs, the handler returns a standard IF\$GET reply message containing status information, as described in Section 4.10.1.9. This function is used to wait for signals such as ring, lost carrier, and so forth.

When this function is requested, the handler 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 waiting process will hang, that is, 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.

## STANDARD DEVICE HANDLERS

**4.10.1.8 Set Status Function** - The Set Status 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.

**4.10.1.9 Get Status Function** - The Get Status function returns information about the interface unit's status. The handler 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.)

**4.10.1.10 Device-Independent Function Modifiers** - If bit FMSBSM of DP.FUN is set, the XL handler signals a binary or a counting semaphore, as described in Sections 4.1.2 and 4.1.3. The setting of bit FMSINH of DP.FUN (inhibit soft-error retry) is not meaningful for serial line service and is ignored by the XL handler.

### 4.10.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a XL handler request (following field DP.SEM) is described below for each type of function.

## STANDARD DEVICE HANDLERS

**4.10.2.1 Block-Mode Read or Write Functions** - The function-dependent portion of a block-mode read request (function code IFSRDP or IFSRDL) or of a write request (function code IFSWTP or IFSWTL) is shown below:

DP.DAD -	(Unused)	Portion of the request sent by value
	(Unused)	
DP.BUF -	Buffer address	
DP.PAR -	-- PAR value --	Portion of the request sent by reference
DP.LEN -	-- Buffer length --	

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 of the data transfer, in bytes.

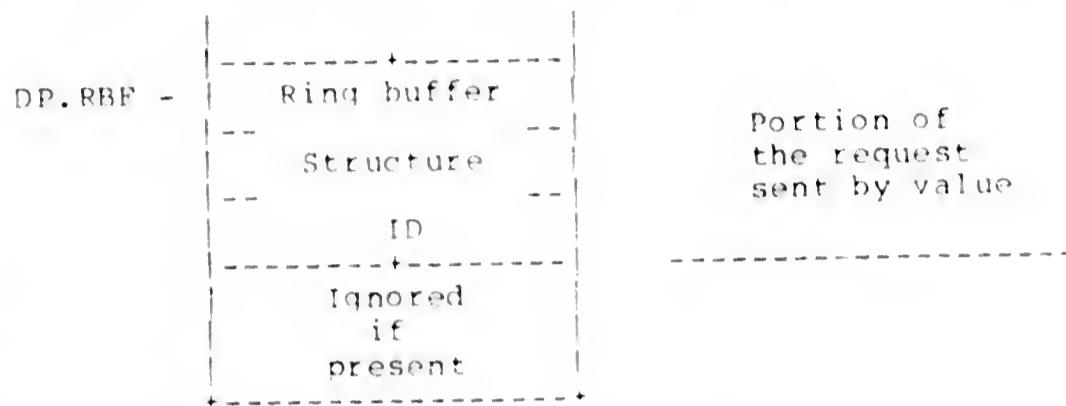
**4.10.2.2 Connect Receive or Transmit Ring Buffer Functions** - The function-dependent portion of a Ring Buffer Connect request for either input or output (function code IFSCRR or IFSCXR) is shown below:

DP.RBF -	Ring buffer	Portion of the request sent by value
	-- Structure --	
	-- ID --	
	-----	
	Ignored if present	

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.

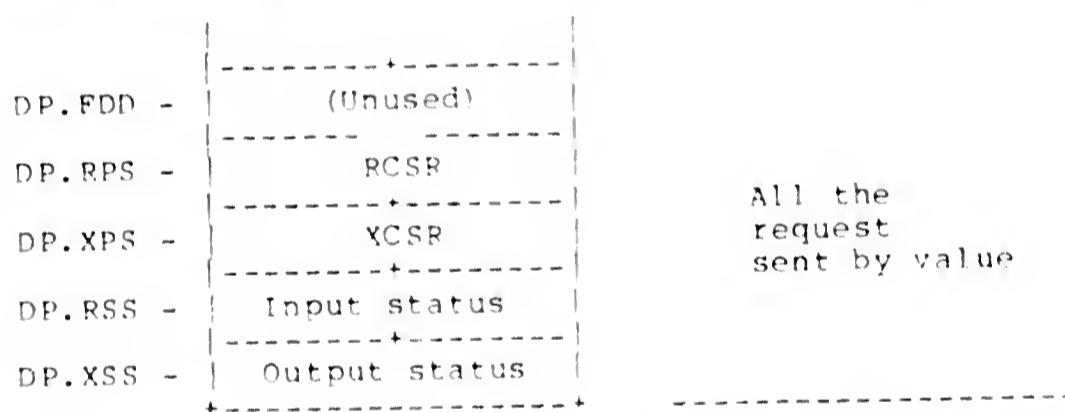
## STANDARD DEVICE HANDLERS

**4.10.2.3 Disconnect Receive or Transmit Ring Buffer Functions** - The function-dependent portion of a Ring Buffer Disconnect request, for either input or output (function code IFSDRR or IFSDXR), 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 handler.

**4.10.2.4 Set Status Function** - The function-dependent portion of a Set Status request (function code IFSSET) 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.

## STANDARD DEVICE HANDLERS

The format of the receiver status-setting word is as follows:

15		0
+-----+-----+		+-----+
x   x   x   x   x   x   x   x   x   x   c   x   x   x   r   e		s   r   t   r
+-----+-----+		+-----+

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 (DLVII-E only).
  - The rx bit (2), if set, indicates request to send (DLVII-E only).
  - The sx bit (3), if set, indicates secondary Xmt (DLVII-E only).
  - The ec bit (5), if set, enables modem control (DLVII-E only).

The format of the transmitter status-setting word for the DLVII, DLVII-E, DLVII-F, DLVII-I, and MXVII-A serial line interfaces 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.
  - 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 MXVII-B and SBC-11/21 serial line interfaces is as follows:

15 g  
+-----+-----+-----+  
| x | x | x | x | x | x | x | x | x | x | b | b | b | b | b | b | t | r | b |  
+-----+-----+

## STANDARD DEVICE HANDLERS

Proceeding from right to left in the format above:

- The sb bit (3), if set, requests a PREAK 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 initiated unit).

The format of the software status-setting words is as follows:

15	0
-----+-----	
1   1   1   x   1   1   1   1	
x   x   x   x   0   x   x   x   x   x   x	
-----+-----	

Proceeding from right to left in the format above:

- The xc bit (1), if set, requests XTN-XORF processing for the input side. No meaning is assigned to this bit for the output side.

4.10.2.5 Get Status Function - The function-dependent portion of a reply to a Get Status request (function code 1FSGET) is shown below:

DP.CLS =	-----    Type   Class  -----	
DP.RPS =	-----   -----  RCSR  -----	
DP.XPS =	-----  XPS  -----	All the request is returned by value
DP.RSS =	-----  Input status  -----	
DP.XSS =	-----  Output status  -----	

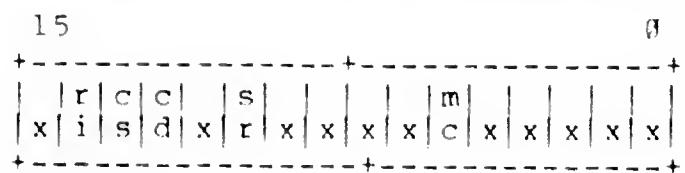
In the information above:

- Class is DCSTER for serial line interfaces
- Type indicates the specific type of DLVII interface, as follows:

TTSPL	Minimum serial line capability (DLVII, DLVII-I, MXVII-A)
TTSDL	DLVII-E
TTSDL	DLVII-F
TTSDLT	MXVII-B, SBC-11/21

## STANDARD DEVICE HANDLERS

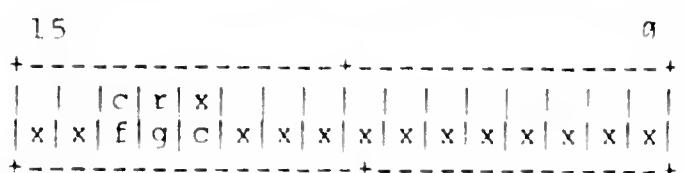
The bit settings of DP.RPS and DP.XPS are used to report the setting of status control bits in the RCSR and XCSR of the specified interface unit. The format of the status-setting word is as follows:



Proceeding from right to left in the 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.

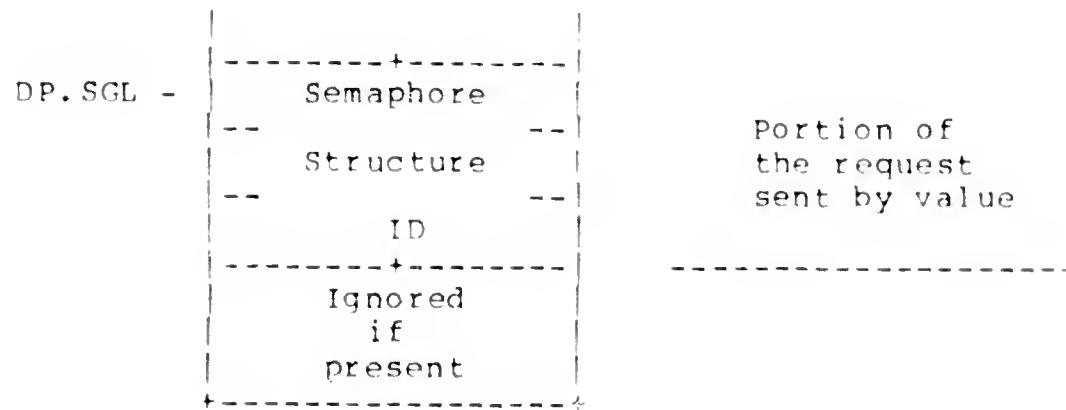
The format of the software status-setting words is as follows:



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 rq 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.

**4.10.2.6 Report Data-Set Status Change Function** - The function-dependent portion of a Report Data-Set Status Change request (function code IFSRSC) is shown below:

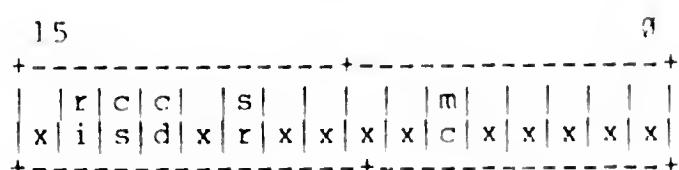


## STANDARD DEVICE HANDLERS

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.

### 4.10.3 Status Codes

The XL handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal success
IS\$FUN	Invalid function code
ISSNXU	Nonexistent unit
ISSOVR	Overrun error on received data
ISSPAR	Parity error on received data

## STANDARD DEVICE HANDLERS

### 4.11 DPV11 COMMUNICATIONS LINE (XP) DEVICE HANDLER

The XP device handler supports X.25 Level II Link Access Procedure Balanced (LAPB) communications, using DPV11 communications interface hardware. X.25, a subset of the High Level Data Link Control (HDLC) standard issued by the International Standards Organization (ISO), permits an application to establish error-free point-to-point communications with other X.25 nodes. The XP handler supports a maximum configuration of 16 DPV11 devices (controllers) per system.

X.25 combines synchronization with message framing in a manner transparent to the process requesting XP handler service. DPV11 hardware parses the bitstream for synchronization characters, eliminating the need for the XP handler to test and count each character.

The synchronous transmission used by the DPV11 is a more bit-efficient method of serial data transmission than the asynchronous method used by devices supported by the XL handler. Neither start nor stop bits are used; instead, special flag characters delimit each message frame. The X.25 message frame format is described in detail in Section 4.11.4.

DPV11 hardware performs a cyclic redundancy check (CRC), eliminating the need for error checking in the XP handler.

The XP handler does not support the Level III packet-switching layer of X.25.

The XP handler consists of three processes -- a user-request process (SXPDRV), a timeout process (SXPTMR), and a main control process (SXPCTL) -- that handle all DPV11 devices in a system. The timeout process, which performs message timeouts and initiates retransmissions as required, requires the use of the CLOCK process (see Chapter 6). The functions performed by this handler are invoked by sending I/O request packets to the handler's request queue semaphore (SXPA). SXPDRV checks the request packet for valid information and then normally dispatches to SXPCTL; SXPCTL performs the requested operation. When the operation is completed, an appropriate completion-status message is returned to the requesting process.

Configuring the DPV11 handler consists of editing a prefix file to establish a correspondence between DPV11 device-handler unit numbers and physical (hardware) device CSR and vector addresses. A maximum of 16 DPV11 devices can be configured in a system application. Additional configuration data, including timeout interval, retry count, and node addresses, can be set during runtime by issuing a Set Characteristics request.

#### 4.11.1 Functions Provided

The functions provided by the DPV11 handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFS\$RDPP (0)	Read physical
IFS\$RLP (1)	Read logical
IFS\$WTP (3)	Write physical
IFS\$WLP (4)	Write logical

## STANDARD DEVICE HANDLERS

IFSSET (6)	Set Line Characteristics
IFSGET (7)	Get Line Characteristics
IFSCON (9)	Connect Link
IFSDSC (10)	Disconnect Link
IFSGMS (12)	Sense Modem Status
IFSSMS (13)	Set Modem Status

For communications link service, there is no distinction between physical and logical function requests. Either function code may be used.

The read function (IF\$RDP or IF\$RDLL) is performed in block mode. The handler reads one information frame -- an unspecified number of characters -- from the communications line into the buffer area identified in field DP.BUF of the request message. Successful completion of the read function ensures valid data; the received frame will have passed a checksum and sequence test, ensuring that the frame is error-free and has arrived in proper sequence, even when transmission errors cause frames to be retransmitted. Unlike the write function, completion of the read function is not required within a specified timeout period.

Information frames received from a remote node when no read request is pending are discarded. However, the XP handler informs the remote node that it is not ready to receive information frames. The remote node should then periodically retry sending the information frame.

The write function (IF\$WTP or IF\$WTLL) is performed in block mode. The handler transmits one information frame -- the contents of the buffer area identified in field DP.BUF of the request message -- to the receiving node. Successful completion of the write function ensures that error-free reception has been acknowledged by the node. If errors occur, the number of retries specified in the Set Status function are attempted before returning an error message to the requesting process.

The Set Characteristics function (IFSSET) permits changing operating parameters either before or after a Connect function has been performed. However, IFSSET should normally be performed only before the Connect request. Parameters that may be set include timeout interval, number of retries, and node addresses; however, the default configuration is for X.25 DTE usage.

The Get Characteristics function (IFSGET) reads, by reference, the communications line statistics into the status buffer specified in the request. The IFSGET function is useful for diagnosing communications network trouble. In addition to the information returned in the status buffer, the device class code is returned in the DP.CLS field of the returned packet. If no buffer is specified, only the device class is returned.

The Connect function performs X.25 link setup on the specified communications line. The DPV11 handler initializes all associated line state variables and performs the necessary exchange of control frames with the remote X.25 node, ensuring that message numbering will be synchronized at both ends of the link.

Since X.25 is basically a symmetrical protocol, the process at either end of the communications link must issue a Connect request in order to establish the communications link. If the communications link does not successfully become established within a specified time interval -- see configuring information below -- the connect will be

## STANDARD DEVICE HANDLERS

signaled as having failed. If, as is usual, one system node is earlier than the other in issuing the Connect function request, the XP handler on that system should periodically issue the Connect function request until the link is established.

The Disconnect function resets the communications line to idle status. This function is most useful on dialup lines, providing a graceful way of shutting down a station.

The Set Modem Status function (IFSSMS) permits direct control over certain modem status bits. X.25 does not specifically mention this function, since the protocol is capable of error detection and recovery. The XP device-handler default values for modem status are sufficient for all X.25 communications. Therefore, the use of this function is not necessary for normal X.25 communications applications; the function is included for special communications applications.

The Sense Modem Status function (IFS\$GMS) permits direct access to modem status bits. X.25 does not specifically mention this function, since the protocol is capable of error detection and recovery. Therefore, the use of this function is not necessary for normal X.25 communications applications; the function is included for special communications applications, such as loopback testing.

### 4.11.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a DPVII handler request (following field DP.SEM) is described below for each type of function.

**4.11.2.1 Block-Mode Read or Write Functions** - The function-dependent portion of a block-mode read request (function code IFSRDP or IFSRDL) or a block-mode write request (function code IFSWTP or IFSWTL) is shown below:

DP.DAD	+-----+   (Unused)   +-----+   (Unused)   Portion of the request +-----+ sent by value
DP.BUF	Buffer address
DP.PAR	+-----+ Portion of the   Par value   request sent +-----+ by reference
DP.LEN	Buffer length

The unit number, in the function-independent portion of the request, selects the desired DPVII controller (one unit per controller); unit numbering starts at 0. The buffer address specifies the destination of the data field in the information frame read in Receive functions or the source buffer containing the data field for an information frame to be transmitted in write functions. The buffer length specifies the buffer size for received messages in read functions and the number of bytes to be transmitted in a frame in write functions.

## STANDARD DEVICE HANDLERS

4.11.2.2 **Connect Link Function** - The Connect function (IF\$CON) request contains no device-dependent function modifiers. Communications line parameters used in the request are default values assumed by the XP handler upon initialization, except as modified by a previously issued Set Status request.

4.11.2.3 **Disconnect Link Function** - The Disconnect function (IF\$DIS) request contains no device-dependent function modifiers.

4.11.2.4 **Set Characteristics Function** - The Set Characteristics function (IF\$SET) permits setting certain communications line parameters. The parameters are passed by value in the function-dependent portion of the request, as shown below:

DP.DAD	+-----+   (Unused)   +-----+
DP.TMO	+-----+   Timeout interval   +-----+
DP.RTY	+-----+   Retry count   +-----+
DP.DTE	+-----+   Local node address   +-----+
DP.DCE	+-----+   Remote node address   +-----+

In the information above:

- Timeout interval is in 0.1-second units (default=0.3 seconds)
- A retry count of 0 implies infinite retries (default=10 retries)
- The XP handler assumes a default value of 3 for DP.DTE and 1 for DP.DCE

### NOTE

The XP handler will discard all frames it receives containing a node address other than that contained in DP.DCE or DP.DTE.

Identical node addresses must be used for DP.DTE and DP.DCE when executing local loopback tests.

4.11.2.5 **Get Characteristics Function** - The function-dependent portion of a Get Characteristics request is shown below:

## STANDARD DEVICE HANDLERS

DP.DAD	(Unused)	Portion of the request sent by value
DP.BUF	Status buffer address	Portion of the request sent by reference
DP.PAR	Par value	Portion of the request sent by reference
DP.LEN	Buffer length	

Status buffer contents are as follows:

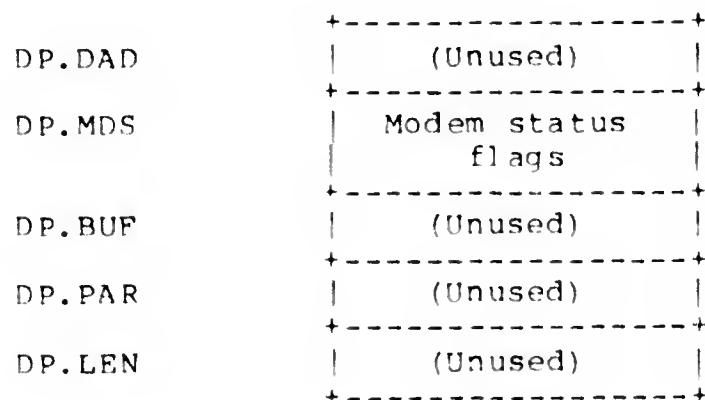
S.FLAG	Status flag bits
S.CTOD	Minute        Second
	Day        Hour
	Year        Month
S.RABT	Receiver abort count
S.ROVR	Receiver overrun error count
S.CRCE	Receiver CRC error count
S.TABT	Transmit abort count
S.TUND	Transmit underrun error count
S.IFMT	Transmitted information frame count
S.SEMT	Transmitted supervisory frame count
S.UEMT	Transmitted unnumbered frame count
S.IFMR	Received information frame count
S.SFMR	Received supervisory frame count
S.UFMR	Received unnumbered frame count

## STANDARD DEVICE HANDLERS

Fields S.IFMT, S.SFMT, S.IFMR, and S.SFMR are double word counts in Pascal LONGINT format. Flag bits in S.FLAG are defined below:

<b>Symbol</b>	<b>Bit</b>	<b>Description</b>
F\$TSAR	1	SARM was transmitted during Connect function
F\$RSAR	2	SARM was received during Connect function
F\$PFBT	3	P/F bit to be sent in next transmitted frame
F\$RBFA	4	User receiver buffer is available
F\$RFRM	5	Error-free frame was received by handler ISR
F\$TBSY	6	Transmitter ISR busy
F\$CONN	7	Information transfer in progress

**4.11.2.6 Set Modem Status** - The function-dependent portion of a Set Modem Status request (sent by value) is shown below:



Modem status flag bits in field DP.MDS are described as follows:

Symbol	Bit	Default Value	Description
RC\$SF	0	0	Select frequency mode (0) or loopback mode (1)
RC\$DTR	1	1	Data set ready
RC\$RTS	2	1	Request to send
RC\$LL	3	0	Local loopback mode

## STANDARD DEVICE HANDLERS

4.11.2.7 Sense Modem Status - The function-dependent portion of a Sense Modem Status request (sent by value) is shown below:

	+-----+-----+	
DP.DAD	Type   Class	
DP.MDS	Modem status flags	
DP.BUF	(Unused)	
DP.PAR	(Unused)	
DP.LEN	(Unused)	
	+-----+-----+	

In the information above:

- Class is DCSCOM, for communications device class
- Type is CMSX25 for DPV11 type

Modem status flag bits in field DP.MDS are described as follows:

Symbol	Bit	Description
RC\$SF	0	Select frequency mode (0) or loopback mode (1)
RC\$DTR	1	Data set ready
RC\$RTS	2	Request to send
RC\$LL	3	Local loopback mode
TX\$SQ	5	Signal quality/test mode in PCSCR
RC\$DSR	9	Data set ready/data mode in RXCSR
RC\$CAR	12	Carrier detected/receiver ready in RXCSR
RC\$CTS	13	Clear to send status in RXCSR
RC\$RNG	14	Ring indicator in RXCSR
RC\$DSC	15	Data set change indicator in RXCSR

### NOTE

For additional information on modem control and DPV11 register bits, refer to the DPV11 Serial Synchronous Interface Technical Manual.

## STANDARD DEVICE HANDLERS

### 4.11.3 Status Codes

The XP device handler returns the following completion-status codes in field DP.STS of the reply message.

Code	Meaning
ISSNOR	Successful completion
IS\$FUN	Invalid function code
IS\$NXU	Nonexistent unit: Unit number specified not supported in system configuration
IS\$OFL	Off line: Line not connected
ISSTM0	Timeout: Connect, Disconnect, Send, or Receive function failed after the number of retries specified in a Set Status function. In the Disconnect function, the disconnect is completed, but the complete sequence involved in the protocol was not completed

For ISSNOR status returns, field DP.ALN of the reply message contains the number of bytes transferred.

### 4.11.4 X.25 Protocol Message Frame Format

X.25 is a bit-oriented protocol in which messages are transmitted in frames. Each frame is bounded by flags consisting of eight bits each -- always a bit sequence of 0111110 -- that delimit the start and end of the frame. When a frame is followed immediately by another frame, the ending flag of the first frame may also be the starting flag of the next frame. In order to prevent data bits from being mistaken for flags, the transmitter hardware automatically inserts a 0 bit following any five consecutive 1s; this is called zero bit insertion or bit stuffing. Receiver hardware automatically removes the 0 immediately following any five consecutive 1s, preserving data integrity. Thus, flags are easily distinguished from data bits and are transparent to both the XP handler and the process requesting communications service.

Figure 4-4 illustrates the X.25 frame format.

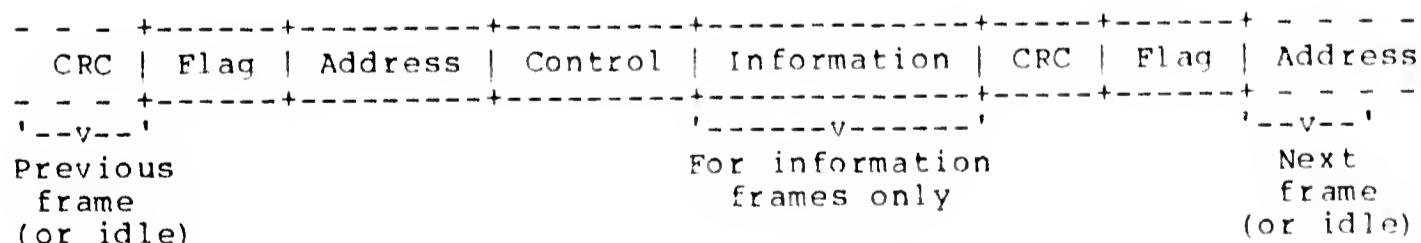


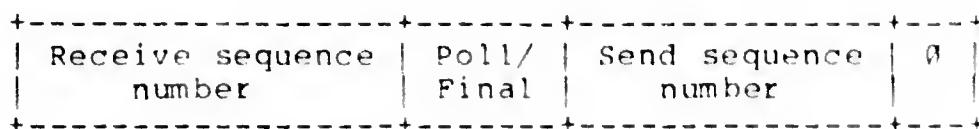
Figure 4-4 General X.25 Frame Format

## STANDARD DEVICE HANDLERS

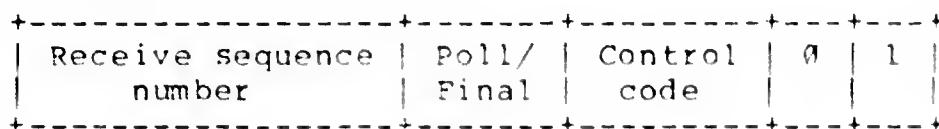
Immediately following the first flag, an 8-bit address indicates the intended message receiver. Although only point-to-point communication between each DPV11 and one X.25 node is supported by the XP device handler, the address field is retained for HDLC compatibility in order to distinguish between node commands and node responses; users normally need not specify an address. A default address value of 3 is used in most secondary node applications for correctly accessing the primary X.25 node. However, in special applications or when the DPV11 is a primary node (address=1), the address value can be changed from the default value to any value in the range 0 to 255(decimal); this can be done by using the Set Characteristics function.

The control byte immediately follows the address byte. This 8-bit value identifies the frame as an information frame, supervisory frame, or nonsequenced frame (also called an unnumbered frame). The three control byte formats are shown in Figure 4-5.

Information frame:



Supervisory frame:



Unnumbered frame:

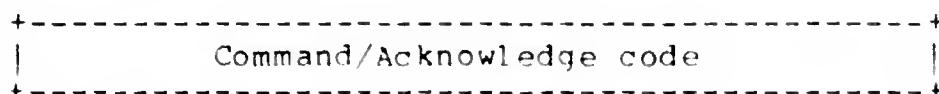


Figure 4-5 Command Byte Formats

Note that bit 0, the first bit transmitted, is always 0 for information frames; a 1 indicates that the frame is either a supervisory frame or an unnumbered frame. In the latter case, bit 1 is 0 for supervisory frames and 1 for unnumbered frames. The significance of the remaining values in the command byte is given below for each frame type:

- Information frames: The command byte contains a 3-bit Send sequence frame count number. Frames are numbered sequentially modulo 8, starting with sequence frame count 0, permitting the receiver node to determine if any frames were lost during message transmission. Numbering is performed automatically by the XP device handler and is transparent to processes requesting communications services.

The control byte for information and supervisory frames contains a Receive sequence frame count number acknowledging the total number of message frames received from the remote node. This information is transparent to the process requesting communications service.

The poll/final bit is used for error recovery. Its use is transparent to processes requesting communications services.

## STANDARD DEVICE HANDLERS

- **Supervisory frames:** Used for link control functions, these frames are short and contain no information field. The supervisory frame includes the 3-bit Receive sequence frame count number and poll/final bit as described for information frames.

The 2-bit command code contains status or error information for this node in response to message frames received, as follows:

Code	Meaning
00	Receiver ready: All sequence frames through the receive sequence number indicated -1 are correct; ready to receive the next frame
01	Receiver not ready: Temporarily busy; cannot accept information frames; however, supervisory or unnumbered frames can be accepted
10	Reject received supervisory frame: Request retransmission

- **Unnumbered frames:** Similar to supervisory frames, but no Receive sequence number is included. Unnumbered frames are used by the XP device handler during link setup and disconnection. All command/acknowledge functions of the unnumbered frame are transparent to the process requesting communications service. Byte values for unnumbered frame command and acknowledge codes are as follows:

Code	Meaning
017	Set asynchronous response mode
103	Disconnect link
143	Unnumbered frame acknowledge

In information frames, the information field immediately follows the command byte. This field is any number of bytes as required for the read or write communications request. All bits are data bits; no control bits are transmitted in this field. The information contained in this field conforms exactly to the data passed in the block I/O operation.

The last field in each frame is the CRC field. This 16-bit checksum is generated by transmitter hardware and is used by receiver hardware for error checking. The CRC value is transparent to both the XP device handler and the process requesting communications service.

## STANDARD DEVICE HANDLERS

### 4.12 DRV11 PARALLEL-LINE (YA) DEVICE HANDLER

The YA device handler supports I/O operations on devices connected through a DRV11 parallel-line interface unit. The handler allows simultaneous input and output on the same unit.

The handler assumes that the DRV11 unit transfers 16 data bits in parallel between itself and its associated devices. That is, the handler moves data to and from the interface unit in word mode. Also, the handler assumes that the REQ A signal represents an output interrupt request and that the REQ B signal represents an input interrupt request; the handler enables the corresponding DRCSR status bits for those purposes.

The handler performs block-mode input and output operations. A specified (even) number of bytes are transferred, by word, to or from the requestor's data buffer per I/O request. The only error indication returned is ISSEFUN, for an invalid function code.

The YA handler consists of one static process and two dynamic ones. The two dynamic processes consist of separate read and write processes. The static process includes the INITIALIZE procedure, which creates the required semaphores, as follows:

SYAA	Queue semaphore for all driver requests
SYAARQ	Queue semaphore for all read operations
SYAAWQ	Queue semaphore for all write operations
SYAAIB	Binary semaphore signaled by read interrupts
SYAAIA	Binary semaphore signaled by write interrupts

The INITIALIZE procedure also creates and starts the two dynamic processes on start-up. The two dynamic processes are as follows:

DRV11_I0	( NAME := 'YAARDR', PRIORITY := 249 )
DRV11_I0	( NAME := 'YAAWRT', PRIORITY := 249 )

After completing its initialization functions at process priority 249, the static process functions as a dispatcher process for I/O requests. The dispatcher process performs request-handling functions for both dynamic processes.

Since the process priority for both dynamic processes is higher than that for the dispatcher, they run until they wait on their respective request queues. Since initially no I/O requests are pending, both dynamic processes lower their process priority to 160, permitting the dispatcher to receive requests as they become issued.

The dispatcher receives requests via a request queue. The request queue semaphore names are SYAA, SYAB, SYAC, and so forth; for example, SYAA is associated with unit A, and SYAB is associated with unit B. Thus, each DRV11 unit is treated as one controller, with no units, for purposes of device selection.

## STANDARD DEVICE HANDLERS

### NOTE

Each DRVII interface unit is factory configured with standard device register and interrupt vector addresses. When more than one DRVII 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, specify the device register and vector addresses used -- whether standard or not -- in the YAPFX.PAS file before building the application software. (Software configuration procedures are discussed in the MicroPower/Pascal-RT System User's Guide and in the MicroPower/Pascal-RSX/VMS System User's Guide.)

When a request is issued to the YA handler, the dispatcher validates the function code. If the function code is valid, the dispatcher sends the request packet to either the YAARDR process, for read requests, or the YAAWRT process, for write requests, as appropriate. Upon receiving request packets, the YAARDR or YAAWRT process transfers the number of words specified in the request packet and sends an appropriate completion status to the requesting process.

If an illegal function code is detected and if a reply queue semaphore was specified in the request, the dispatcher returns a reply packet to the sender with an illegal function (ISSEFUN) status. If a binary or a counting semaphore was specified for replies, the dispatcher signals the specified semaphore.

The YAARDR (read) process effects the block-mode transfer of data from the DRVII to the specified user buffer each time a data word is available in the DRVII's input buffer register. The YA handler executes interrupt-driven transfers; that is, the read process waits on the SYAAIB (read interrupt request) binary semaphore until the DRVII is ready to input a new data word. When ready for the data transfer, the DRVII issues an interrupt request. The kernel responds to the interrupt request by signaling the SYAAIB binary semaphore, and the read process transfers the next word from the DRVII's input buffer register to the specified buffer. Input transfers continue in this manner until the specified number of words have been input. Once the block-mode transfer has been completed, the handler returns the appropriate completion status to the requesting process.

Similarly, output transfers are effected by the YAAWRT (write) process through interrupt-driven transfers. The handler waits on the SYAAIA binary semaphore until the DRVII is ready to output a data word. When ready for the data transfer, the DRVII requests an interrupt. The kernel responds to the interrupt by signaling the SYAAIA semaphore, and the write process transfers the next word from the specified write buffer to the DRVII's output data register. Output transfers continue in this fashion until the specified number of words have been output. Once the block-mode transfer has been completed, the handler returns the appropriate completion status to the requesting process.

## STANDARD DEVICE HANDLERS

### 4.12.1 Functions Provided

The functions provided by the YA handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IF\$RD P (0)	Read physical
IF\$RL P (1)	Read logical (equivalent to read physical)
IF\$WT P (3)	Write physical
IF\$WT L (4)	Write logical (equivalent to write physical)
IF\$GET (7)	Get Characteristics

For parallel-line service, there is no distinction between physical and logical function requests; either function code may be used.

Both read and write requests transfer only an even number of bytes to or from the user's buffer; data is transferred by word. Therefore, if an odd buffer length is specified in the request (field DP.LEN), the handler assumes length-1 bytes as the effective transfer length.

**4.12.1.1 Get Characteristics Function** - The Get Characteristics function returns, in the function-dependent portion of the reply message, a block of device-dependent information. The information consists of device class, type, and subtype.

**4.12.1.2 Device-Dependent Function Modifiers** - The YA handler does not recognize any device-dependent function modifiers. The values of modifier bits 6 to 12 of the function word (DP.FUN) are not significant.

**4.12.1.3 Device-Independent Function Modifiers** - If bit 13 of DP.FUN is set, the YA handler sends a full reply to the requestor, as described in Sections 4.1.2 and 4.1.3. The setting of bit 15 of DP.FUN (inhibit soft-error retry) is not meaningful for parallel-line service and is ignored by the YA handler.

### 4.12.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a YA handler request (following field DP.SEM) is described below for each type of function.

## STANDARD DEVICE HANDLERS

4.12.2.1 **Read or Write Functions** - The function-dependent portion of a read or write request (function code IFSRDP, IFSRDL, IF\$WTP, or IFSWTL) is shown below:

DP.DAD -	Not used	Portion of the request sent by value
DP.BUF -	Buffer address	Portion of the request sent by reference
DP.PAR -	PAR value	
DP.LEN -	Buffer length	

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 of the data transfer, in bytes. If the length value is an odd number, the last byte is not transferred.

4.12.2.2 **Get Characteristics Function** - The Get Characteristics request returns the following information in the function-dependent portion of the reply packet:

DP.FDD -	Type   Class
	Subtype

In the information above:

- Class is DCSRLT, for real-time device class
- Type is RT\$DRV, for DRV11
- Subtype is RSSPRT, for parallel device subtype

### 4.12.3 Status Codes

The YA handler returns the following completion-status codes in field DP.STS of the reply message.

Code	Meaning
ISSNOR	Successful completion; read or write block transfer completed
ISSFUN	Invalid function code

For ISSNOR status returns, field DP.ALN of the reply message contains the number of bytes transferred.

No information is returned in field DP.ERR of the reply message.

## STANDARD DEVICE HANDLERS

### 4.13 SBC-11/21 PARALLEL PORT (YF) DEVICE HANDLER

The YF device handler supports block I/O operations on devices connected through the 8255-type parallel I/O (PIO) ports on the SBC-11/21 (FALCON or FALCON-PLUS) module. The handler supports unidirectional input and/or output with 8-bit transfers as selected by jumpers on the module. Since both PIO ports are contained on the SBC-11/21 module, only one controller is supported. Each of the two ports is considered a separate unit within the controller.

The YF handler 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. (Refer to the SBC-11/21 Single-Board Computer User's Guide for a complete description of PIO modes and other hardware configuration considerations.)

The YF handler prefix file YFPFX.MAC 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. Each SBC-11/21 interface unit is factory configured with standard device register and interrupt vector addresses.

The handler performs block-mode input and output operations. A specified number of bytes are transferred to or from the requestor's data buffer per I/O request. Physical and logical I/O requests are executed in exactly the same manner. Once the requested transfer has been completed, the handler returns a status code that indicates either normal completion (ISSNOR) or an error condition (ISSINV, ISSABT, or ISSNXU).

The YF handler consists of one request-handling main process and an interrupt service routine (ISR) common to read and write function requests. The handler contains separate internal queues for each unit. Each unit has a data area that indicates the I/O direction. The main process has a request queue on which it receives requests for operations on the PIO units. The request queue semaphore name is SYFA.

## STANDARD DEVICE HANDLERS

### NOTE

As an alternative to the YF handler interface described in this section, DIGITAL provides three routines for setting up and accessing the SBC-11/21 PIO ports directly from a user process, using noninterrupt mode (programmed I/O). These routines -- SET PIO MODE, WRITE PIO, and READ PIO -- and the type definitions necessary for their use, are supplied in the Pascal source files YFDRV.PAS and YFDRV1.PAS. YFDRV.PAS, which defines the routines, must be compiled and then merged with the program at user-process build time; YFDRV1.PAS must be included in the user program at compile time. Both files contain comments describing the use of the interface they provide.

#### 4.13.1 Functions Provided

The functions provided by the YF handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSLRP (1)	Read logical (equivalent to read physical)
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical (equivalent to write physical)
IFSGET (7)	Get Characteristics

For PIO operations, there is no distinction between physical and logical function requests; either function code may be used. Both read and write requests transfer the specified number of bytes to or from the user's buffer.

**4.13.1.1 Device-Dependent Function Modifiers** - The YF handler does not recognize any device-dependent function modifiers; the values of modifier bits 6 to 12 of the function word (DP.FUN) are not significant.

**4.13.1.2 Device-Independent Function Modifiers** - If bit 13 of DP.FUN is set, the YF handler sends a full reply to the requestor, as described in Sections 4.1.2 and 4.1.3. The setting of bit 15 of DP.FUN (inhibit soft-error retry) is not meaningful for PIO service and is ignored by the YF handler.

## STANDARD DEVICE HANDLERS

### 4.13.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a YF handler request (following field DP.SEM) is described below for each type of function.

#### 4.13.2.1 Read or Write Functions - The function-dependent portion of a read or a write request (function code IFSRDP, IFSRDL, IFSWTP, or IFSWTL) is shown below:

DP.SGL -	-----+-----    (Unused)    -----+-----    (Unused)    -----+-----    (Unused)    -----+-----	Portion of the request sent by value
DP.BUF -	Buffer address    -----+-----	-----
DP.PAR -	PAR value    -----+-----	Portion of the request sent by reference
DP.LEN -	Buffer length    -----+-----	-----

The unit number selects the desired I/O port; valid unit numbers are 0 or 1 (port A or port B, respectively). 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 of the data transfer in bytes.

#### 4.13.2.2 Get Characteristics Function - The Get Characteristics request returns the following information in the function-dependent portion of the reply packet:

DP.FDD -	-----+-----    Type   Class    -----+-----      Subtype    -----+-----
----------	--

In the information above:

- Class is DCSRLL, for real-time device class
- Type is RTSFAL, for SBC-11/21 PIO type
- Subtype is RSSPRT, for parallel device subtype

## STANDARD DEVICE HANDLERS

### 4.13.3 Status Codes

The YF handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Successful completion; block transmission completed
ISSABT	Handler process deleted; request not serviced
ISSFUN	Invalid function code; unsupported function or attempt to perform a supported function on a PIO unit not supporting that function (for example, attempting to perform a write operation on a PIO port configured for input transfers)
ISSNXU	Nonexistent unit; unit number greater than 1 specified

For ISSNOR status returns, field DP.ALN of the reply message contains the number of bytes transferred.

No information is returned in field DP.ERR of the reply message.

## STANDARD DEVICE HANDLERS

### 4.14 KXT11-C TUS8 (DD) DEVICE HANDLER

The DD device handler for the KXT11-C peripheral processor is identical to the DD handler for non-KXT11-C processors (described in Section 4.3), except that it includes support for the multivibrator chip that resides on the KXT11-C. Thus, a TUS8 subsystem can be connected to any of the serial lines on the KXT11-C.

Both the KXT11-C and non-KXT11-C versions of the DD handler support logical and physical I/O operations on a TUS8 cartridge tape subsystem. Both handlers also support read-with-increased-threshold and write-verify options and report the storage capacity of the device in terms of logical blocks. See Section 4.3 for detailed descriptions of the functions provided by the DD handlers.

The KXT11-C DD handler resides in the KXT11-C driver library, DRVK.OBJ. To use the KXT11-C DD handler, you must edit its prefix file, DDPFXX.MAC, and then use the prefix file to build the KXT11-C DD handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

## STANDARD DEVICE HANDLERS

### 4.15 KXT11-C TWO-PORT RAM (KK) HANDLER

The KXT11-C is a peripheral processor for LSI-11-bus-based systems. It functions in a master/slave relationship with an arbiter processor running on the LSI-11 bus. The arbiter (master) controls all interactions between the two processors. The KXT11-C (slave) waits for a command from the arbiter processor before initiating any message transfer. The mechanism of arbiter/KXT11-C interactions is a simple request-reply protocol operating through the dual-port registers of the KXT11-C. That protocol is supported by the KK handler, which runs on the KXT11-C, and the KX handler (see the KX handler section), which runs on the arbiter.

#### NOTE

As an alternative to interfacing directly to the KK handler as described in this section, DIGITAL supplies two MicroPower/Pascal functions that support the KXT11-C side of the arbiter/KXT11-C protocol: `KK_write_data` and `KK_read_data`. See Appendix H for detailed descriptions of those functions.

An arbiter/KXT11-C transaction occurs as follows: First, the arbiter interrupts the KXT11-C by issuing a command. By issuing the command, the arbiter causes ownership of the dual-port registers to switch to the KXT11-C. The KXT11-C then reads the command, checks its validity, and acts on the command. After acting on the command, the KXT11-C indicates -- by relinquishing control of the dual-port registers and, if interrupts are enabled, by issuing an interrupt to the arbiter -- that the command has been processed.

The KK and KX handlers implement the protocol via read and write commands that they issue to each other. Each KK read or write operation transfers data between a KXT11-C buffer and the arbiter.

#### NOTE

See Appendix A of the KXT11-C Peripheral Processor Software User's Guide for a detailed description of the KX/KK communication protocol.

The KK handler communicates with the arbiter via the command and status registers of Data Channel 0 (4-byte data area) and Data Channel 1 (12-byte data area) of the KXT11-C dual-port RAM. The KK handler manages the two user-control areas of the KXT11-C dual-port RAM as separate units, numbered 0 and 1, respectively.

In the process of transferring data between a local buffer on the KXT11-C and the LSI-11 bus, the KK handler calls one of two kernel-defined debug locations -- `SKXTQW` or `SKXTQR`. If you are debugging a KXT11-C application with the MicroPower/Pascal symbolic debugger, PASDBG, you can use PASDBG's SET BREAK command to set a breakpoint on one or both of the locations. `SKXTQW` is called just before data is transferred from a channel to a KXT11-C local buffer (a KX handler write operation). `SKXTQR` is called after data has been transferred from a local buffer to a channel but before the LSI-11 bus

## STANDARD DEVICE HANDLERS

is interrupted (a KK handler read operation). Setting breakpoints on either or both of those locations allows you to examine a segment of a message while a read/write operation is suspended in midexecution.

The KK handler resides in the KXT11-C driver library, DRVK.OBJ. To use the KK handler, you must edit its prefix file, KKPFXK.MAC, and then use the prefix file to build the handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

### 4.15.1 Functions Provided

The functions provided by the KK handler are listed below, by symbolic and decimal function code.

Code	Function Performed
IF\$RD P (0)	Read physical
IF\$RD L (1)	Read logical (equivalent to read physical)
IF\$WT P (3)	Write physical
IF\$WT L (4)	Write logical (equivalent to write physical)
IF\$GET (7)	Get Characteristics

The read functions (IF\$RD P and IF\$RD L) instruct the KK handler to get data from the LSI-11 bus. The immediate action caused by the request is the setting of a "data requested" bit and the interrupting of the LSI-11 bus, if bus interruption is enabled. The read request is queued and completes when the arbiter transfers data across the LSI-11 bus to satisfy the request. The data is placed in a user-specified data area.

The write functions (IF\$WT P and IF\$WT L) instruct the KK handler to move data from a user-specified area across the LSI-11 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 Get Characteristics function returns, in the function-dependent portion of the reply message, bit settings that indicate the device class and the device type of the KK handler.

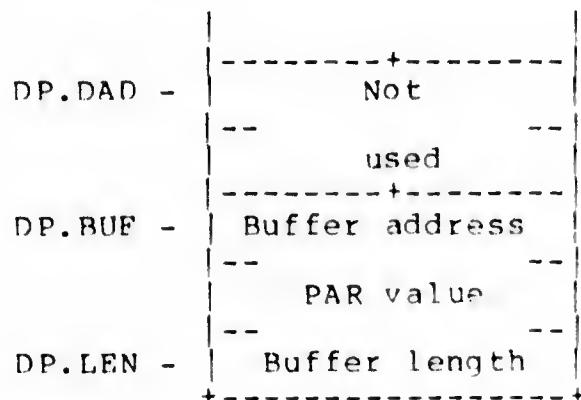
**4.15.1.1 Device-Independent Function Modifiers** - If modifier FM\$BSM (bit 13) of DP.FUN is set, the KK handler signals a binary or counting semaphore, as described in Sections 4.1.2 and 4.1.3.

### 4.15.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of a KK handler request, following field DP.SEM, is described below for each type of function.

## STANDARD DEVICE HANDLERS

4.15.2.1 **Read or Write Functions** - The function-dependent portion of a read or write request -- function code IFSRDP, IFSRDL, IFSWTP, or IFSWTL -- is shown below:



4.15.2.2 **Get Characteristics Function** - The Get Characteristics request returns the following information in the function-dependent portion of the reply packet:



In the information above:

- Class is DC\$PRL, for protocol device class
- Type is PR\$KKK, for KXT11-C two-port-RAM handler

### 4.15.3 Status Codes

The KK handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal completion
ISSABT	Handler aborted; KXT11-C in stand-alone mode or other error
IS\$FUN	Invalid function code
IS\$OVR	Buffer overflow: receiving buffer overflowed before end of message was encountered

## STANDARD DEVICE HANDLERS

### 4.16 KXT11-C DMA TRANSFER CONTROLLER (QD) HANDLER

The QD device handler supports the 2-channel DMA Transfer Controller (DTC) on the KXT11-C Peripheral Processor. The QD handler provides a high-speed method for moving data from place to place. Usually the QD device handler moves data between local storage areas on the KXT11-C and LSI-11 bus locations.

Requests to the QD handler for physical read/write operations specify a DMA address and a local buffer. The DMA addresses used in QD handler operations may be local memory addresses, I/O page addresses, or LSI-11-bus addresses.

#### NOTE

As an alternative to interfacing directly to the QD handler as described in this section, DIGITAL supplies six MicroPower/Pascal functions that support the DTC: SDMA\_TRANSFER, SDMA\_SEARCH, SDMA\_SEARCH\_TRANSFER, SDMA\_ALLOCATE, SDMA\_DEALLOCATE, and SDMA\_GET\_STATUS. See Appendix H for detailed descriptions of those functions.

The QD handler resides in the KXT11-C driver library, DRVK.OBJ. To use the QD handler, you must edit its prefix file, QDPFXK.MAC, and then use the prefix file to build the QD handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

#### 4.16.1 Functions Provided

The functions provided by the QD handler are listed below, by symbolic and decimal function code.

Code	Function Performed
IF\$RD P (0)	Read physical
IF\$WTP (3)	Write physical
IF\$GET (7)	Get Characteristics
IF\$ALL (8)	Allocate Channel
IF\$DEA (9)	Deallocate Channel

Read and write operations can transfer data, search data for a user-specified value, or transfer data while searching, depending on the function modifier specified (see Section 4.16.1.1). Transfer operations copy data from a DMA address to a user-specified buffer (read) or from a user-specified buffer to a DMA address (write). Search operations use a DMA address as the starting point for the search.

A Get Characteristics operation returns the contents of device registers to a user-specified buffer.

## STANDARD DEVICE HANDLERS

An Allocate Channel operation dedicates a channel of the DTC for the use of a single process. Deallocate Channel reverses the effect of the Allocate Channel request.

The QD handler functions are described in more detail in Section 4.16.2.

**4.16.1.1 Device-Dependent Function Modifiers** - A read or a write operation may be modified by the following device-dependent function modifiers:

BIT	Meaning
FMSTTO	Transfer data to/from DMA address without searching the data (the default)
FMSTSO	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 reply packet (DP.ALN)
FMSTTS	Transfer and search data until either a search value is matched or a byte count (DP.LEN) expires; return count of bytes transferred in reply packet (DP.ALN)

**4.16.1.2 Device-Independent Function Modifiers** - If bit FMSBSM is set, the QD handler signals a binary or counting semaphore, as described in Sections 4.1.2 and 4.1.3.

### 4.16.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The QD handler uses the function-independent format described in Section 4.1.2 but incorporates several device-specific functions and modifier bits within that format. In addition to the device-specific functions and modifiers listed in Section 4.16.1, DMA-address modifiers may also be specified in read and write requests. See the section on read and write function formats, below, for a description of the address modifiers.

The desired DMA channel is selected by the unit number (DP.UNI) in the function-independent portion of the request packet. Each QD controller has two DMA channels, numbered unit 0 and unit 1, respectively.

The function-dependent portion of a QD handler request, following the field DP.SEM, is described below for each type of function.

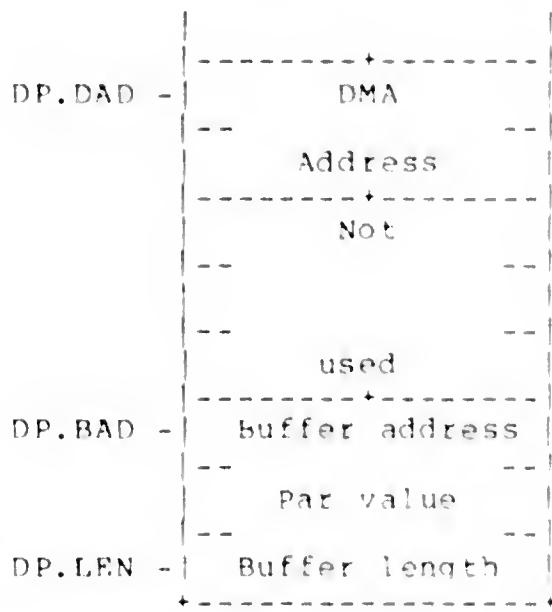
**4.16.2.1 Read and Write Functions** - The read physical (IFSRDP) function directs the QD handler to cause the DTC to transfer data to a KXT11-C buffer. You specify the address from which the data block will be taken, the length of the transfer, the destination, and the DTC channel number (DP.UNI).

## STANDARD DEVICE HANDLERS

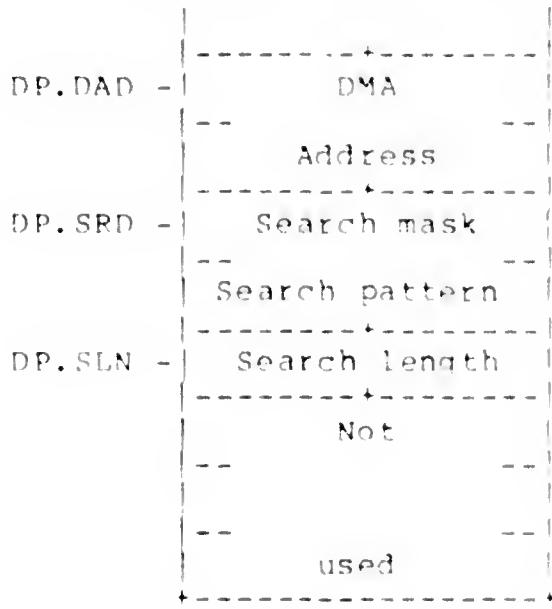
The write physical (IFSWTP) function directs the QD handler to cause the DTC to transfer data from your buffer. You specify the address where the data block will be sent, the length of the transfer, the source buffer location, and the DTC channel number (DP.UNIT).

The read and write physical functions may be modified by the function-dependent bits FMSTTO (the default), FMSTSO, or FMSTTS. Those modifier bits are described in Section 4.16.1.1. In addition, the method of accessing the DMA address specified in a read or a write request is modified by bits in the DMA address field. Those address-modifier bits are described below.

The function-dependent portion of a read or write physical request, following field DP.SEM, is shown below:



If the FMSTSO function modifier is specified, the device-dependent portion of the request takes this form:

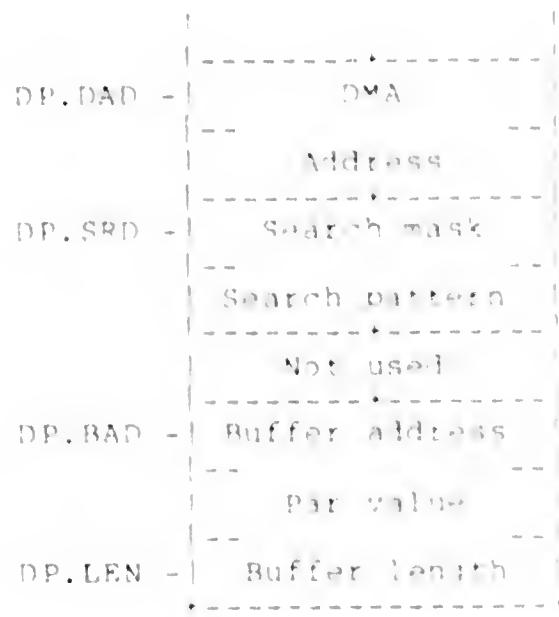


Bits set to 1 in the search mask field (offset DP.SRD) mask out bits in the object word. For example, to search only the low-order byte of each word in a buffer, you should specify a search mask with the high-order eight bits all set to 1. Thus, the low-order byte of each word in the buffer will be compared with the low-order byte of the search pattern.

## STANDARD DEVICE HANDLERS

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 searching the high-order byte, the search pattern must be shifted to the high-order byte.

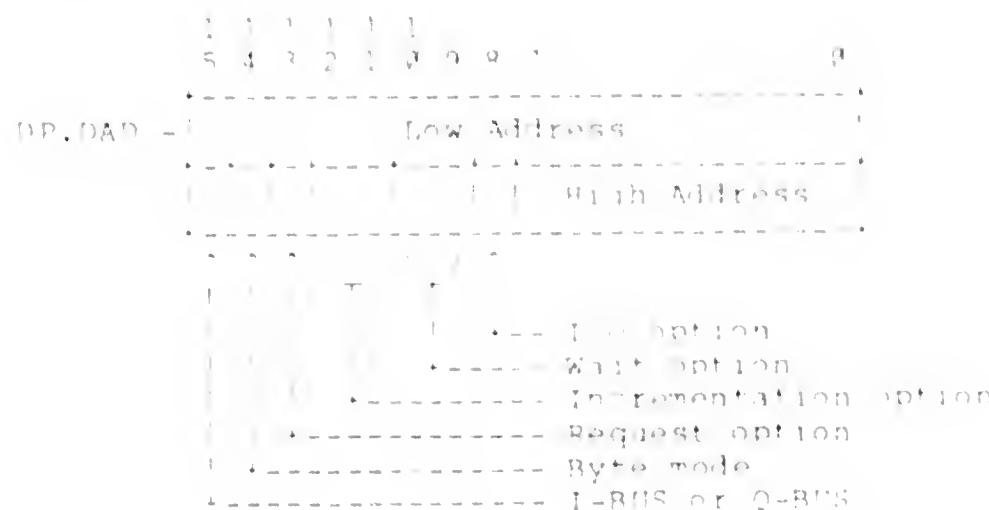
If function modifier FMSTTS is specified, the function-dependent portion of the request takes this form:



The request packet fields shown above have the following significance:

### Field      Significance

**DP.DAD**      The DMA address for the operation. Operations move data either from a user buffer to the DMA address area or from the DMA address area to a user buffer. The user buffer is always in local memory in a ZXT11-11. The DMA address is a 22-bit physical address with addressing control bits. The address may be a local memory address, an I/O page address, or an LSI-11 bus address. The structure of the DP.DAD field is shown below.



## STANDARD DEVICE HANDLERS

### NOTE

The address-modifier bits and fields described below (AM\$xxx) are defined by the kernel macro QDDEF\$ from the KXTDFU.MAC library (part of COMU.SML/.MLB).

As noted in Section 4.1, other symbols used in this chapter to describe the packets and the information they contain (IFSxxx, FMSxxx, DP.xxx, ISSxxx) are defined by the kernel macro DRVDEF\$ from the DRVDEF.MAC library.

#### • I/O option

AMSTIO Transfer to/from I/O page address -- causes DMA address to reference the I/O page as opposed to normal memory locations

#### • Increment options

AMSTIA Increment DMA Address -- causes DMA address to be incremented during the transfer

AMSTDIA Decrement DMA Address -- causes DMA address to be decremented during the transfer

AMSTHA Hold DMA Address -- causes DMA address to be held constant

#### • Wait options

### NOTE

You should specify only one of the wait options (0, 1, 2, or 4).

AMSWSA 0 Wait States -- add no wait states to DMA address access

AMSWSI 1 Wait State -- adds one wait state to DMA address access

AMSWS2 2 Wait States -- adds two wait states to DMA address access

AMSWS4 4 Wait States -- adds four wait states to DMA address access

#### • Request option

AMSWFR Waits for request for transfer to/from DMA address

#### • Byte-mode option

AMSBMO Byte mode to/from DMA address

## STANDARD DEVICE HANDLERS

### NOTE

Byte mode is supported only in local-to-local transfers. If this option is not specified, word-mode operations will be performed.

#### • Local Bus or LSI-11 Bus

AM\$TQB      LSI-11 bus DMA address -- causes DMA addresses to be mapped into the LSI-11 bus space; otherwise, the internal bus is accessed. This option must not be selected if the AMSBMO option is selected.

DP.SLN      Maximum length of the search to be performed, in bytes

DP.BAD      Address of your buffer

DP.LEN      Length of your buffer

### NOTE

The reply packet returned for the read and write functions is the same as described in Section 4.1.3. However, if the function modifier FM\$TSO was specified, the field at offset DP.ALN will contain the count of bytes searched instead of the count of bytes transferred.

**4.16.2.2 Get Characteristics Function** - The IF\$GET function directs the QD handler to return the contents of a specified unit's device registers to a user-specified buffer. The function-dependent portion of a Get Characteristics request, following field DP.SEM, is shown below:

DP.DAD -	+-----+ --   --   Not --   --   used --   -----+-----+
DP.BAD -	Buffer address -----+-----+ --   --   Par value --   -----+-----+
DP.LEN -	Buffer length -----+-----+

## STANDARD DEVICE HANDLERS

Up to 92 bytes -- 46 words -- of status information can be returned by the QD handler. If you specify a buffer length smaller than 92, the handler will return only the number of bytes you request. The format of the status information that is returned is shown below. See the KXT11-CA User's Guide hardware manual for detailed descriptions of the listed registers and counts.

0	Current address req, offset, B, ch. 1
1	Current address req, offset, B, ch. 0
2	Base address req, offset, B, ch. 1
3	Base address req, offset, B, ch. 0
4	Current address req, offset, A, ch. 1
5	Current address req, offset, A, ch. 0
6	Base address req, offset, A, ch. 1
7	Base address req, offset, A, ch. 0
8	Current address req, seg/tag, B, ch. 1
9	Current address req, seg/tag, B, ch. 0
10	Base address req, seg/tag, B, ch. 1
11	Base address req, seg/tag, B, ch. 0
12	Current address req, seg/tag, A, ch. 1
13	Current address req, seg/tag, A, ch. 0
14	Base address req, seg/tag, A, ch. 1
15	Base address req, seg/tag, A, ch. 0
16	Chain load address req, offset, ch. 1
17	Chain load address req, offset, ch. 0
18	Chain load address req, seg/tag, ch. 1
19	Chain load address req, seg/tag, ch. 0
20	Interrupt save register, ch. 1
21	Interrupt save register, ch. 0
22	Command/Status register, ch. 1
23	Command/Status register, ch. 0
24	Current operation count, ch. 1
25	Current operation count, ch. 0
26	Base operation count, ch. 1
27	Base operation count, ch. 0

## STANDARD DEVICE HANDLERS

28	Master mode register
29	Not used
35	Not used
36	Pattern register, ch. 1
37	Pattern register, ch. 0
38	Mask register, ch. 1
39	Mask register, ch. 0
40	Channel mode register, low, ch. 1
41	Channel mode register, low, ch. 0
42	Channel mode register, high, ch. 1
43	Channel mode register, high, ch. 0
44	Interrupt vector register, ch. 1
45	Interrupt vector register, ch. 0

**4.16.2.3 Allocate Channel Function** - The IFSALL function directs the QD handler to reject any future requests that are issued by tasks other than the one that issued the IFSALL request. You supply only the function-independent portion of the request packet, including the DTC channel number (DP.UNI). No function modifiers apply to this function. If any are supplied, they are ignored.

**4.16.2.4 Deallocate Channel Function** - The IFSDEA function directs the QD handler to reverse the effect of a previous Allocate Channel request. You supply only the function-independent portion of the request packet, including the DTC channel number (DP.UNI). No function modifiers apply to this function. If any are supplied, they are ignored.

### 4.16.3 Status Codes

The QD handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal operation
ISSABT	I/O abort
ISSDAL	Device allocated
ISSFUN	Invalid function code

**STANDARD DEVICE HANDLERS**

ISSIVM	Invalid mode
ISSNXM	Nonexistent or read-only memory
ISSNXU	Nonexistent unit

## STANDARD DEVICE HANDLERS

### 4.17 KXT11-C ASYNCHRONOUS SERIAL LINE (XL) HANDLER

The KXT11-C XL device handler supports asynchronous I/O operations on devices connected to any of the three serial I/O ports on the KXT11-C Peripheral Processor. This handler is identical to the (non-KXT11-C) XL handler described in Section 4.10, except that it includes support for the multiprotocol chip that resides on the KXT11-C. Thus, the handler can concurrently service up to three serial ports on the KXT11-C.

#### NOTE

The three serial ports on the KXT11-C can be used by the KXT11-C TU58 (DD) handler as well as by the XL handler. See the two DD handler sections of this chapter for descriptions of the DD handler interface.

In addition, one of the KXT11-C's serial lines -- the multiprotocol chip "A" port -- can be used for synchronous serial I/O via the XS device handler. See the XS handler section of this chapter for a description of the XS handler interface.

The first serial I/O port on the KXT11-C 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 handler 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 requestor's buffer space. In block-mode output, a specified number of bytes are transferred from the requestor's data buffer to the serial line unit.

In ring-mode input, the handler continuously transmits bytes from the serial line unit to an input ring buffer specified by the requestor. 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 handler continuously transmits bytes from an output ring buffer, as they become available, to the specified serial line unit. The requestor 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 handler provides optional X-ON/X-OFF control character processing. If requested to do so, the handler inhibits the output side of a given serial line unit when it detects an X-OFF character on the input side of the same channel. Output resumes when an X-ON is subsequently received.

For the port that supports modem control, the handler allows you to enable data-set interrupts and to receive data-set status information when such interrupts occur. The handler allows you to get data-set status information and to set certain XCSP and RCSR bits. By this

## STANDARD DEVICE HANDLERS

mechanism you can control baud rates, enable interrupts if a modem control signal changes, set modem control signals, and transmit break signals.

The handler'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 the processor must be specified in the KXT11-C XL handler prefix file, XLPFXX.MAC. That file is edited and processed during the building of the application system. (Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.)

### 4.17.1 Functions Provided

The functions provided by the KXT11-C XL handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical (equivalent to read physical)
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical (equivalent to write physical)
IFSSFT (6)	Set Status
IFSGET (7)	Get Status
IFSCRR (8)	Connect Receive Ring Buffer
IFSCXR (9)	Connect Transmit Ring Buffer
IFSDRR (10)	Disconnect Receive Ring Buffer
IFSDXR (11)	Disconnect Transmit Ring Buffer
IFSRSC (12)	Report Data-Set Status Change

**4.17.1.1 Read Function** - The read function (code IFSRDP or IFERDL) is performed in block mode. The handler 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.

**4.17.1.2 Write Function** - The write function (code IFSWTP or IFSWTL) 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 FMSKCF (bit 11) in the function code word (offset DP.FUN) is used to enable or disable automatic X-ON/X-OFF processing. If the bit is set, the handler intercepts any X-OFF character from the input side of the line and disables output for the same channel until a subsequent X-ON character is received. Multiple successive X-ON or X-OFF characters have the same effect as just one such character. If X-ON/X-OFF processing is not selected, the handler passes all input characters to the requestor. The X-ON/X-OFF function

## STANDARD DEVICE HANDLERS

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.

**4.17.1.3 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.

**4.17.1.4 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 handler prefix file.

**4.17.1.5 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 FMSXCK is used to enable or disable automatic X-ON/X-OFF processing as previously described in the write function section.

**4.17.1.6 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.

**4.17.1.7 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 X-ON/X-OFF processing or allows X-ON/X-OFF processing to be selected via the function modifier bit in the write function.

## STANDARD DEVICE HANDLERS

**4.17.1.8 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.)

**4.17.1.9 Report Data-Set Status Change Function** - The Report Data-Set Status Change function (code IF\$RSC) allows the requestor to wait for a change of data-set (modem) status. When a change occurs, the handler returns a standard Get Status reply, as described in the Get Status sections. When this function is requested, the handler 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.

**4.17.1.10 Device-Independent Function Modifiers** - If modifier FMSBSM (bit 13) of DP.FUN is set, the KXT11-C XL handler signals a binary or counting semaphore, as described in Sections 4.1.2 and 4.1.3.

### 4.17.2 Function-Dependent Request Formats

The function-independent portion of a handler request message is described in Section 4.1.2.

#### 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 handler request, following field DP.SEM, is described below for each type of function.

## STANDARD DEVICE HANDLERS

**4.17.2.1 Block-Mode Read or Write Functions** - The function-dependent portion of a block-mode read or write request is shown below:

DP.DAD -	Not -- used --	Portion of the request sent by value
DP.BUF -	Buffer address -- PAR value --	Portion of the request sent by reference
DP.LEN -	Buffer length	

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 of the data transfer, in bytes. The PAR value is filled in by the operating system.

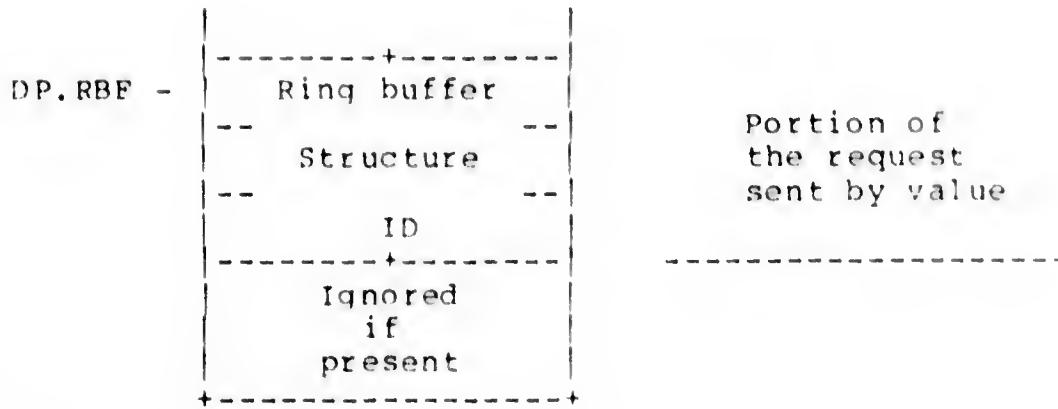
**4.17.2.2 Connect Receive or Transmit Ring Buffer Functions** - The function-dependent portion of a Ring Buffer Connect request for either input or output is shown below:

DP.RBF -	Ring buffer -- Structure -- -- ID --	Portion of the request sent by value
	Ignored if present	

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.

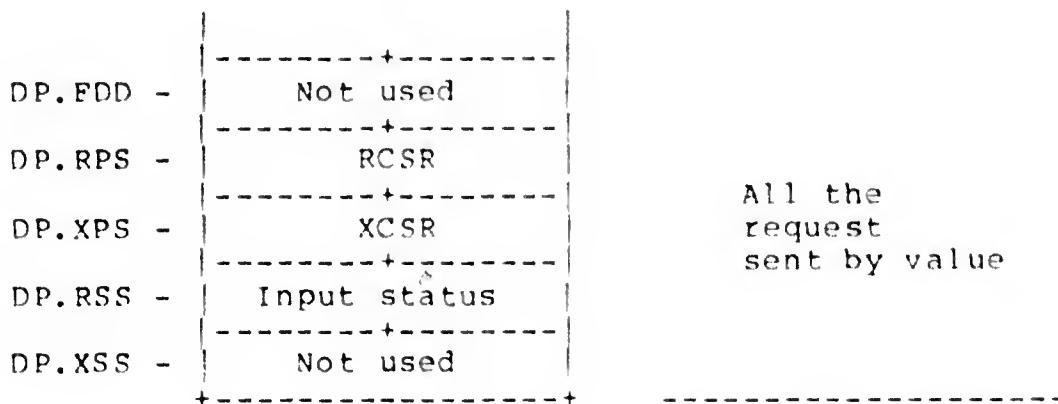
**4.17.2.3 Disconnect Receive or Transmit Ring Buffer Functions** - The function-dependent portion of a Ring Buffer Disconnect request for either input or output is shown below:

## STANDARD DEVICE HANDLERS



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 handler.

**4.17.2.4 Set Status Function** - The function-dependent portion of a Set Status request (code IFSET) is shown below:



The request packet fields shown above have the following significance:

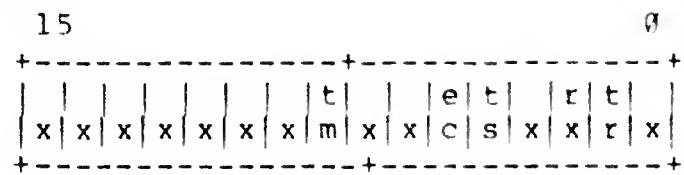
Field	Significance
-------	--------------

DP.RPS      Status control bits to be set in the 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:

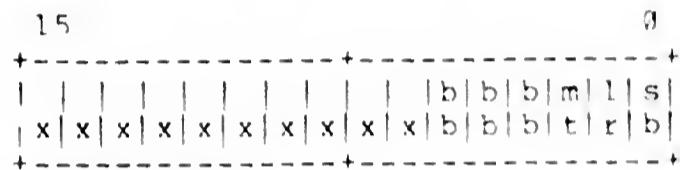


## STANDARD DEVICE HANDLERS

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.

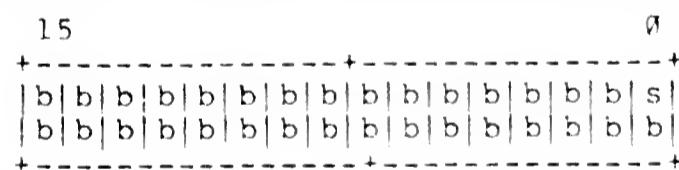
The format of the transmitter status-setting word (offset DP.XPS) for the DLART device is as follows:



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).

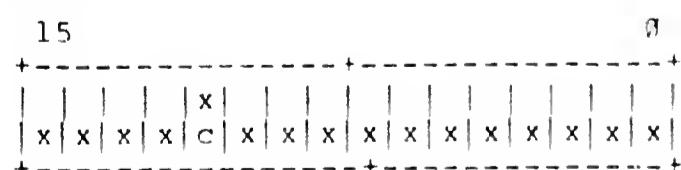
The format of the transmitter status-setting word (offset DP.XPS) for a multiprotocol device is as follows:



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 bb bits (1 to 15) indicate the desired baud rate. See the KXT11-CA User's Guide hardware manual for values.

The format of the software status-setting word (offset DP.RSS) is as follows:



Proceeding from right to left in the format above:

## STANDARD DEVICE HANDLERS

- The xc bit (11), if set, enables X-ON/X-OFF processing regardless of the state of the X-ON/X-OFF function modifier bit within the block write or Connect Transmit Ring Buffer commands (see the write function description).

4.17.2.5 Get Status Function - The function-dependent portion of a reply to a Get Status request (function code IFSGET) is shown below:

DP.CLS	Type	Class
DP.RPS	RCSR	
DP.XPS	XCSR	All the request is returned by value
DP.RSS	Input status	
DP.XSS	Output status	

In the information above:

- Class is DC\$TER, indicating a serial line
- Type indicates the specific type of serial line device, as follows:

TTSDLT	Serial line with programmable baud rate, exception indicator flags, and a self-test mode -- the DLART port on the KXT11C.
TTSDM	First or second port on the multiprotocol chip on the KXT11C when used in the asynchronous data-leads-only mode. The port then supplies the functionality of the TTSDLT type device.
TTSDMM	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 handler or no handler is in control of the second multiprotocol port.
TTSDMP	Multiprotocol port operating in the asynchronous mode with partial modem control. Only the first port on the multiprotocol device can be this device type. A TTSDMP device is the same as the TTSDMM device except that only the Receiver Active and Clear to Send modem input signals are supported. This device type allows another device handler besides XL to use 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

## STANDARD DEVICE HANDLERS

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:

15	0
-----+-----+	
l   c   r   d     t     t   t   t   t	
x   c   s   r   m   x   x   m   x   x   c   s   x   x   r   x	
-----+-----+	

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 (ATTN MM - VME 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 (ATTN MM - VME only).

The format of the transmitter status word (offset DP.XPS) for the DLART device is as follows:

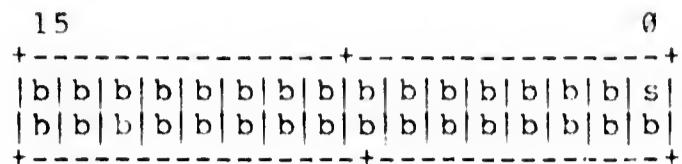
15	0
-----+-----+	
b   b   b   b   m   s	
x   x   x   x   x   x   x   x   x   x   b   b   b   b   t   r   b	
-----+-----+	

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 PBRF 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).

## STANDARD DEVICE HANDLERS

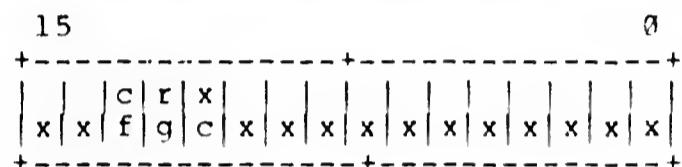
The format of the transmitter status word (offset DP.XPS) for a multiprotocol device is as follows:



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. See the KXT11-CA User's Guide hardware manual for values.

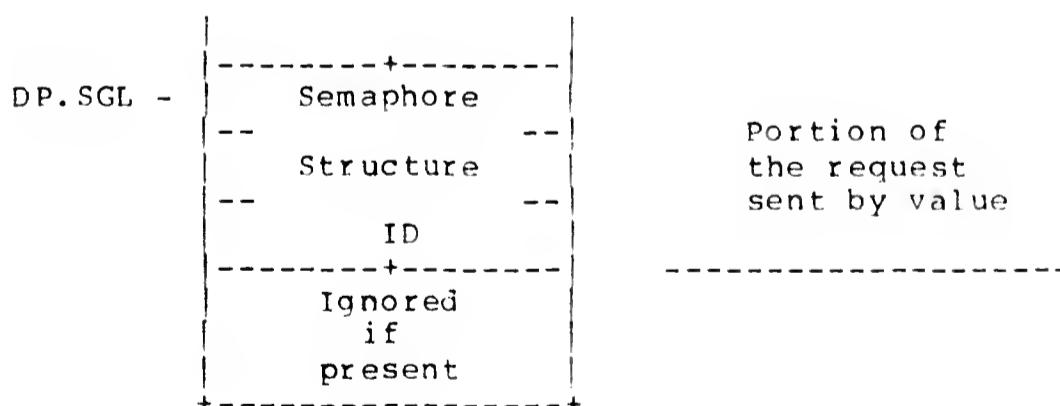
The formats of the receiver and transmitter software status words (offsets DP.RSS and DP.XSS) are identical, as follows:



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.
  - The cf bit (13), if set, indicates that the port was connected to a ring buffer during configuration.

**4.17.2.6 Report Data-Set Status Change Function -** The function-dependent portion of a Report Data-Set Status Change request (function code IFSRSC) 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.

## STANDARD DEVICE HANDLERS

### 4.17.3 Status Codes

The XL handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Normal success
IS\$FUN	Invalid function code
ISSNXU	Nonexistent unit
ISSOVR	Overrun exception on received data
ISSPAR	Parity exception on received data

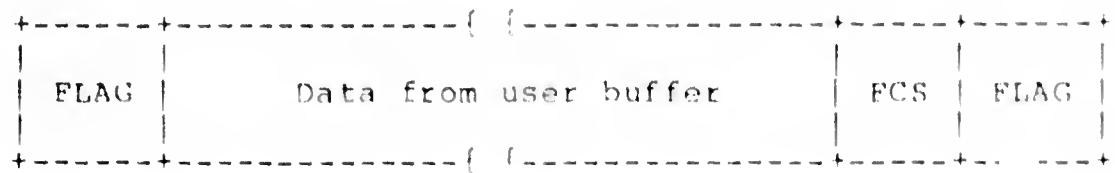
## STANDARD DEVICE HANDLERS

### 4.18 KXT11-C SYNCHRONOUS SERIAL LINE (XS) HANDLER

The XS device handler supports synchronous serial I/O via the KXT11-C multiprotocol chip SLU2 Channel A, allowing you to establish an elementary bit-oriented communications channel. The handler performs the following functions of bit-oriented communications procedures: synchronization (flag detection), transparency (bit stuffing), invalid frame detection, frame abortion detection, and Frame Check Sequence (FCS) checking/calculation. The handler can be used by user-written software as a component in performing such bit-oriented communication procedures as CCITT X.25, ISO HDLC, IBM SDLC, ANSI ADCCP, and others. However, the XS handler does not, in itself, completely perform any of those procedures.

The XS handler provides no explicit modem control. However, the line enable and line disable services do control modem signals as necessary to activate and deactivate a modem.

The XS handler provides basic HDLC style framing and error detection.



Data sent by the handler has the FCS (Frame Check sequence) appended to form a frame. If any errors are detected as the handler sends the frame, it is aborted and resent. Frames received by the handler 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.
- If, after a period of time, the originator of a frame determines that no acknowledgment for that frame has been received, it resends the frame.

For further information about the techniques of data communication, refer to the Technical Aspects of Data Communication (John E. McNamara, Digital Press, 1982) or a book of your choice on data communication principles.

The XS handler runs as three processes in a KXT11-C application: an interface process that receives and validates user requests and queues them internally; a receiver process that handles the incoming flow of data and sends it directly to the user; and a transmitter process that takes data from the internal queues and transmits it to the device.

## STANDARD DEVICE HANDLERS

The XS handler makes no assumptions about the configuration of the multiprotocol chip, other than assuming that interrupts have been enabled for the device.

The XS handler resides in the KXT11-C driver library, DRVK.OBJ. To use the XS handler, you must edit its prefix file, XSPFXK.PAS, and then use the prefix file to build the XS handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

The source files for the XS handler are provided on the MicroPower/Pascal distribution kit, to assist you in writing your own handler for the KXT11-C multiprotocol chip.

### 4.18.1 Functions Provided

The functions provided by the XS handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Get Frame
IFSWTP (3)	Send Frame
IFSENA (8)	Enable Line
IFSDSA (9)	Disable Line
IFSSTP (10)	Kill Requests

The Get Frame function causes a frame of data to be passed from the device to a user-specified buffer.

The Send Frame function causes a user-supplied buffer of data to be transmitted to the device.

The Enable Line function turns a line on and readies it for communication.

The Disable Line function turns a line off and disconnects it from any communication.

The Kill Requests function aborts all Send Frame and/or all Get Frame service in progress and returns Send Frame and/or Get Frame requests from the handler's internal queues in the order they were submitted to the handler.

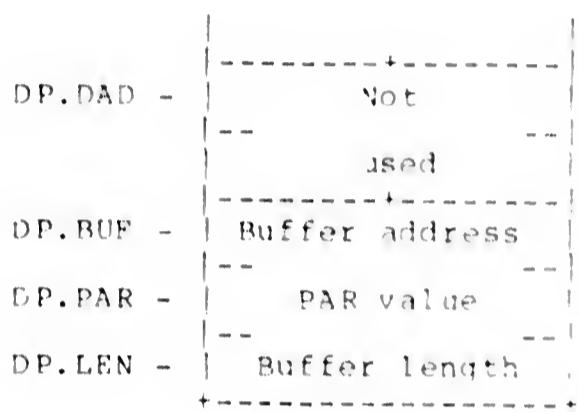
Because the XS handler performs the bit-oriented communications functions mentioned above -- flag detection, bit stuffing, invalid frame detection, frame abortion detection, and Frame Check Sequence (FCS) checking/calculation -- any frame returned by a Get Frame request is a valid frame with a correct FCS and with all bit stuffing removed. Correspondingly, frames transmitted by the Send Frame request have appropriate bit stuffing and FCS added by the XS handler.

## **STANDARD DEVICE HANDLERS**

#### 4.18.2 Function-Dependent Request Formats

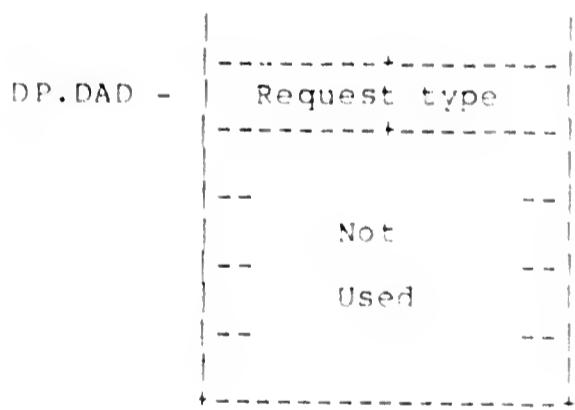
The function-independent portion of a handler request message is described in Section 4.1.2. The function-dependent portion of an XS handler request, following field DP.SEM, is described below for the Get Frame, Send Frame, and Kill Requests functions. The function-dependent portion of the request message is not used for the Enable Line and Disable Line functions.

**4.18.2.1 Get Frame and Send Frame Functions** - The function-dependent portion of a Get Frame or Send Frame request is shown below:



The fields at offsets DP.BUF and DP.LEN, above, give the location and length of the user's buffer.

**4.18.2.2 Kill Requests Function** - The function-dependent portion of a Kill Requests packet is shown below:



The request-type field, at offset DP.DAD, indicates the type(s) of requests to be aborted. A value of 1 kills all Send Frame requests, a value of 2 kills all Get Frame requests, and a value of 3 kills both Send Frame and Get Frame requests.

## STANDARD DEVICE HANDLERS

### 4.18.3 Status Codes

The XS handler returns the following completion-status codes in field DPSTS of the reply message:

Code	Meaning
ISSNOR	Normal completion
IS\$FUN	Invalid I/O function code
ISSOVF	Overflow
ISSSTP	I/O processing stopped

## STANDARD DEVICE HANDLERS

### 4.19 KXT11-C PARALLEL PORT AND TIMER/COUNTER (YK) HANDLER

The YK device handler supports the parallel I/O and programmable timer/counter hardware on the KXT11-C Peripheral Processor. Among the functions the handler provides are parallel I/O (PPI) read, PPI write, pattern recognition, DMA read, DMA write, and counter/timer set, read, and clear.

#### NOTE

As an recommended alternative to interfacing directly to the YK handler as described in this section, MC68020 supplies six MicroPower/Pascal functions that support the PPI and timer/counter hardware on the KXT11-C: YK PORT READ, YK PORT WRITE, YK SET PATTERN, YK SET TIMER, YK READ TIMER, and YK CLEAR TIMER. See Appendix H for detailed descriptions of these functions.

The YK handler resides in the KXT11-C driver library, KXVK.DBL. To use the YK handler, you must edit its prefix file, KXPYK.MAC, and then use the prefix file to build the YK handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

The YK handler provides an interface with two independent 8-bit parallel ports, a 4-bit special-purpose I/O port, and three independent 16-bit counter/timers. The following options are available when you configure the 8-bit parallel ports:

- Port can be input, output, bidirectional, or mixed operation mode. Mixed operation occurs when 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-set bit 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.

## STANDARD DEVICE HANDLERS

With interlocked handshake, the handler 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. Or, 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 handler 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 independent 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 handler. A pulsed handshake is used to interface to mechanical-type devices. It is not available to bidirectional ports.

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 handlers; 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. The 3-wire handshake is not available in the bidirectional mode. However, because the direction of the port can be changed under software control, bidirectional IEEE-488-type transfers can be performed.

- 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 handler 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 handler 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.

## STANDARD DEVICE HANDLERS

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, l-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 l-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 the counter mode, the I/O line of the port associated with the counter/timer is used as an external counter input.

In gate mode, the I/O line of the port associated with the counter/timer is used as an external gate input to the counter/timer.

In trigger mode, the I/O line of the port associated with the counter/timer 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.

## STANDARD DEVICE HANDLERS

### 4.19.1 Functions Provided

The functions provided by the YK handler are listed below, by symbolic and decimal function code:

Code	Function Performed
IFSRDP (0)	Read physical
IFSRDL (1)	Read logical (equivalent to read physical)
IFSRDV (2)	Read virtual (equivalent to read physical)
IFSWTP (3)	Write physical
IFSWTL (4)	Write logical (equivalent to write physical)
IFSWTV (5)	Write virtual (equivalent to write physical)
IFSGET (7)	Get Characteristics
IFSYKP (8)	Set Pattern
IFSYKR (9)	DMA Read
IFSYKW (10)	DMA Write
IFSYKE (11)	DMA Complete
IFSYKS (12)	Set Timer
IFSYKU (13)	Clear Timer
IFSYKT (14)	Read Timer

The physical, logical, and virtual read functions transfer data from a parallel port to a KXT11-C buffer. If pattern-match mode is set, the transfer terminates when a user-specified pattern is found or a search limit is reached; otherwise, you specify the length of the transfer.

The physical, logical, and virtual write functions transfer data from a KXT11-C buffer to a parallel port. As with the read functions, if pattern-match mode is set, the transfer terminates when a user-specified pattern is found or a search limit is reached; otherwise, you specify the length of the transfer.

The Get Characteristics function returns, in the function-dependent portion of the reply message, bit settings that indicate the device class and the device type of a specified parallel port or timer unit.

The Set Pattern function sets pattern-match mode on parallel port A or B.

The DMA functions allow you to perform DMA transfers via the parallel ports. To perform a DMA transfer, you first send a DMA Read or DMA Write request to the YK handler and wait for a reply. If the reply indicates normal status, you then send a DMA transfer command to the QD handler -- see the QD handler section of this chapter -- and wait for the request to complete. After the DMA transfer completes, you send a DMA Complete request to the YK handler. The DMA Complete request unlocks the request queue for the port that was used for the transfer.

The timer functions control the timer/counters on the KXT11-C. The Set Timer function can initialize a timer constant, trigger the timer after setup, or set up a timer to periodically signal a binary semaphore. The Read Timer function returns the current count value from a timer, and the Clear Timer function deactivates a timer.

## STANDARD DEVICE HANDLERS

**4.19.1.1 Device-Dependent Function Modifiers** - The following device-dependent function modifiers are used in conjunction with YK handler requests:

Code	Bit	Function Performed
FMSYKW	11	Sets wait-for-pattern-match mode (Set Pattern only)
FMSYKO	10	OR pattern mode -- match with any bit specified in pattern argument (Set Pattern only)
FMSYKA	9	AND pattern mode -- all bits specified in pattern argument must match (Set Pattern only)
FMSYKC	11	Continuous cycle -- cause timer to signal a binary semaphore and restart after each timeout (Set Timer only)
FMSYKI	9	Initialize timer constant value (Set Timer only)
FMSYKT	8	Trigger timer after setup (Set Timer only)
FMSYKR	8	Reset pattern mode -- reset a previously set pattern mode at the end of a read or a write (Port A or B Read/Write)

If none of the three Set Timer modifiers is specified, the timer mode set by the prefix file or by the last timer command remains in effect.

Read and write operations are affected by the three Set Pattern modifiers listed above. In pattern-match mode, read and write operations terminate either when a user-specified pattern is found or when a user-specified byte count (DP.LEN) expires. If no pattern modifiers are specified in a Set Pattern request, the default is FMSYKA (all bits must match).

### 4.19.2 Function-Dependent Formats

The function-independent portion of a handler request message is described in Section 4.1.2. The following unit numbers may be specified at offset DP.UNI in the function-independent portion of a YK handler request:

Code	Unit
YK.A (0)	Port A
YK.B (1)	Port B
YK.C (2)	Port C
YK.1 (3)	Timer 1
YK.2 (4)	Timer 2
YK.3 (5)	Timer 3

The function-dependent portion of a YK handler request, following field DP.SEM, is described below for each type of function.

## STANDARD DEVICE HANDLERS

4.19.2.1 **Read Functions** - The function-dependent portion of a read function request packet is shown below:

DP.DAD	-----+----- Not --          -- used -----+-----
DP.BUF	Buffer address -----+----- --          -- PAR value -----+-----
DP.LEN	Buffer length -----+-----

4.19.2.2 **Write Functions** - The function-dependent portion of a write function request packet is shown below:

DP.PDA	-----+----- Bufferless data -----+----- Not used -----+-----
DP.BUF	Buffer address -----+----- --          -- PAR value -----+-----
DP.LEN	Buffer length -----+-----

If you are writing a single byte to a port -- or 2 bytes to a linked port -- you can omit the buffer specification and place the data in the DP.PDA field.

4.19.2.3 **Get Characteristics Function** - The Get Characteristics request returns the following information in the function-dependent portion of the reply packet:

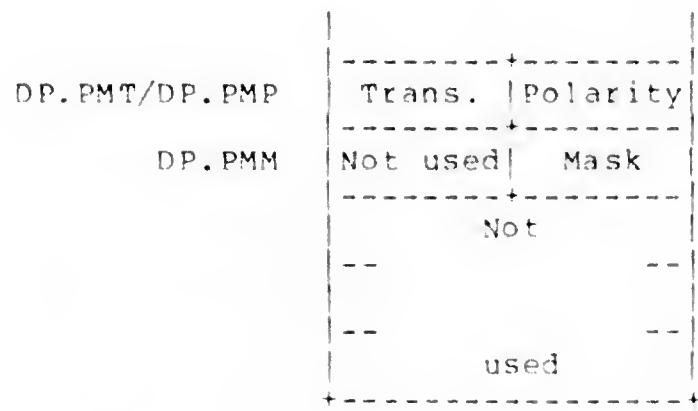
DP.FDD	- Type   Class -----+-----
--------	-------------------------------

In the information above:

- Class is DC\$RLT, for real time device class
- Type is RTSYKP, for KXT11-C parallel port, or RTSYKT, for KXT11-C timer

## STANDARD DEVICE HANDLERS

4.19.2.4 Pattern Set Function - The function-dependent portion of a Set Pattern function request packet is 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, or 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 FMSYKA).

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 DE.PMP, 10000000 in DP.PMT, and 11101111 in DP.PMM.

## STANDARD DEVICE HANDLERS

4.19.2.5 Timer Set Function - The function-dependent portion of the Timer Set function request packet is shown below:

DP.PDA	Timer constant
DP.TSM	Signal
--	semaphore
--	ID
--	Not used

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.

4.19.2.6 Timer Read Function - The function-dependent portion of the Timer Read function request packet is shown below:

DP.PDA	Timer pointer
--	Not
--	used
--	
--	

The DP.PDA field contains a pointer to the variable that is to receive the timer count value. If this field is omitted, the timer count value will be returned in the first word of the function-dependent portion of the reply packet, as follows:

DP.FDD	Timer count
--	

## STANDARD DEVICE HANDLERS

### 4.19.3 Status Codes

The YK handler returns the following completion-status codes in field DP.STS of the reply message:

Code	Meaning
ISSNOR	Successful operation
IS\$FUN	Illegal function code in request
ISSIVM	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, other port must be bit type
ISSNXU	Nonexistent unit number specified
ISSOVE	Buffer size was exceeded before a PIO pattern match was detected

## CHAPTER 5

### MICROPOWER/PASCAL FILE SYSTEM

This chapter describes the DIGITAL-supplied subroutines, processes, and macros available for doing block-structured input and output using the MACRO-11 language. Appendix F is a sample program that uses the routines described in this chapter.

#### 5.1 MACRO-11 FILE INPUT AND OUTPUT

MACRO-11 interfaces to the MicroPower/Pascal file system through the \$QIO subroutine, which the QIOS macro request is predefined to call. You can use the \$QIO subroutine to perform directory operations and block-structured I/O.

The directory service process, named RTDSP, performs all directory operations for an application. RTDSP creates and maintains directories in RT-11 format; thus, MicroPower/Pascal files are compatible with RT-11 file utility programs.

RTDSP also limits the access that other concurrent processes have to an open file. When a file is opened, RTDSP keeps the file specification on an active file list until the file is closed. Files on the active file list can be accessed by processes other than the opening process only in the manner specified when RTDSP first opened the file.

\$QIO uses a data area called a file identification block (FIB) for each open file. The FIB\$ macro has been predefined to declare a data area for use as a file identification block.

The file specification can be placed in a file identification block by using the \$PARSE subroutine. This routine parses MicroPower/Pascal file specifications and places the parsed file specification at a specified address in the form used in file identification blocks. The PARSE\$ macro calls \$PARSE.

#### 5.2 \$QIO SUBROUTINE

The \$QIO subroutine supports only I/O using blocked buffering. By definition, blocked buffering is arranged such that input and output to the device occur one buffer of data at a time; the entire buffer is read from or written to the device. The size of a blocked buffer is normally made an integral number of blocks (512 bytes) -- the unit of data handled by device driver processes for block-structured devices.

## MICROPOWER/PASCAL FILE SYSTEM

The devices to which block-structured I/O is done may or may not be file-structured. For example, although a terminal is not file-structured, it can be used for block-structured input and output by using a blocked buffer.

\$QIO recognizes a group of predefined functions that request directory operations by the directory service process or input and output operations by a driver process. These predefined functions are discussed in the sections below.

To perform directory operations, \$QIO is used to send request messages to RTDSP; to perform I/O operations, \$QIO is used to send request messages to a device driver process. The priority assigned to these requests equals the value assigned to the symbol QIO\$PR. For example, to assign a priority of 50 to \$QIO requests, you would enter the following statement in the symbol definition section of your program:

```
QIO$PR == 50
```

If an error occurs during execution of a \$QIO function or if an illegal function is requested, an error code is returned in the FB.CST field of the file identification block in use. Chapter 4 of the MicroPower/Pascal-RT Messages Manual describes these error codes. Also, \$QIO will return with the carry bit set if an error occurs.

### 5.2.1 \$QIO Read and Write Functions

For all read and write operations, the FB.DVR field of the file identification block for the open file must contain the structure ID of the device driver request queue semaphore. The address of the data buffer must reside in the FB.BFP field, and the size of that buffer, in bytes, must be placed in the FB.BFZ field. Individual read and write requests may require additional information, as described below.

Virtual, logical, and physical reads and writes may be requested from \$QIO. However, you should not mix these types of operations when interacting with an open file.

Read and write operations include:

- IF\$WRV
- IF\$RDV
- IF\$WRL
- IF\$RDL
- IF\$WTP
- IF\$RDP

5.2.1.1 Virtual Reads and Writes - These are the read and write functions normally used for input and output to files on devices with directories. Virtual read and write operations are performed on a block of data that is offset from the beginning of the file. To perform a virtual operation, \$QIO calculates the block number to be accessed next and transfers the number of bytes indicated in the FB.BFZ field between the device and the buffer for each request.

## MICROPOWER/PASCAL FILE SYSTEM

For virtual operations, the logical starting block of the file must be in the FB.SBK field of the file identification block in use, and the size of that file must be in the FB.FSZ field. These fields are filled by a successful request to \$QIO to open the file. \$QIO requires the virtual block number in the FB.BNO field.

\$QIO returns the SEOF bit of the FB.STA set if the end of file is encountered. The carry bit is set if a partial read or write occurs.

Code	Function
IF\$WRV	Requests a virtual write
IF\$RDV	Requests a virtual read

**5.2.1.2 Logical Reads and Writes** - The value in the FB.SBK field of the file identification block in use is taken as the logical block to be accessed. To read or write sequential blocks using logical operations, you must increment the value in FP.SBK by the number of blocks transferred after each I/O operation.

No end-of-file checking is performed.

Code	Function
IF\$WRL	Requests a logical write
IF\$RDL	Requests a logical read

**5.2.1.3 Physical Reads and Writes** - For these operations, data is transferred between the data buffer and the address on the device (see Chapter 4) specified in the FB.SBK field of the file identification block.

No end-of-file checking is performed.

Code	Function
IF\$WRP	Requests a physical write
IF\$RDP	Requests a physical read

**5.2.1.4 Function Modifiers** - These function modifiers affect the way in which reads and writes are performed. They do not pertain to directory operations. Function modifiers are passed directly to the device driver process and must, therefore, be recognized by that process (see Chapter 4).

Code	Function
FMSWFM	Requests formatting of the floppy disk in the unit specified by the FB.DVC, FB.CTL, and FB.UNI fields of a file identification block. This modifier must be specified in conjunction with IF\$WTP, making the complete function IF\$WTP+FMSWFM. If modifier FMSWSD is also set, the floppy disk is formatted for single density; otherwise, it is formatted for double density.

## MICROPOWER/PASCAL FILE SYSTEM

FM\$WSD	Requests single-density formatting of the floppy disk. This modifier must be specified in conjunction with FM\$WFM and IFSWTP.
FM\$BSM	Indicates that the reply semaphore is a binary semaphore.
FM\$INH	Inhibits retries by device driver processes for devices that have a storage medium.

### 5.2.2 \$QIO Directory Operations

To manipulate the directory of a device, you can use \$QIO to send a request to RTDSP. Directory operations include:

- IF\$CLS IF\$DEL IF\$ENT IF\$PRG
- IF\$LKP IF\$REN IF\$PRO IF\$UNP
- IF\$INI IF\$DIR

#### 5.2.2.1 File-Opening Functions - Files can be opened on both file-structured and non-file-structured devices.

How other processes can share an open file depends on the setting of the \$RW and \$SHARE bits in the FB.STA field of the file identification block when the file is initially opened. If \$SHARE is set, other processes can read from the file. When \$RW is set, other processes can also write to the file. If another process subsequently attempts to open a file, the same type of access must be specified; otherwise, an error is returned to that process. The application process must synchronize access to shared files.

When a file is opened, the logical starting block of the file is returned in the FB.SBK field of the file identification block. The file size assigned is returned in the FB.FSZ field. For non-file-structured devices, such as terminals, \$QIO returns the value 0 in both the FB.SBK and FB.FSZ fields. For file-structured devices without directories, such as an RX02 being used as a single file, the value 0 is returned in the FB.SBK field; the size of the device, in blocks, is returned in FB.FSZ.

Code	Function
IF\$ENT	Requests that space be obtained on the device and that a tentative directory entry be created with the name specified in the file identification block. The size of the free space obtained depends on the value in the FB.FSZ field. A 0 in FB.FSZ obtains half the largest space on the device; -1 obtains the largest. A nonzero positive number obtains that number of blocks. A value less than -1 results in an error.
IF\$LKP	Requests that RTDSP look up an existing file on the device. The size of the retrieved file is placed in the FB.FSZ field of the specified file identification block. The logical block number on which the file begins is placed in the FB.SBK field.

## MICROPOWER/PASCAL FILE SYSTEM

**5.2.2.2 File-Closing Functions** - When a file is closed, one instance of that file specification is removed from the active file list in RTDSP. If several processes have opened a file, the file remains open to processes that have not closed it; processes open and close shared files independently. For directory-structured devices, the value in FB.SBK is subtracted from the value in FB.HBK and is entered on the directory as the file size.

Code	Function
IF\$CLS	Requests that RTDSP close the file and make it permanent.
IF\$PRG	Requests that RTDSP close the file but not make it permanent. However, if the file was previously permanent, it will remain permanent.

**5.2.2.3 File Utility Functions** - For any of these functions, a file identification block containing a file or device specification, a device driver structure ID, and a reply semaphore ID must be supplied.

Code	Function
IF\$PRO	Protects the file designated in the specified file identification block from being deleted. The file to be protected must not be open; otherwise, an error occurs, and the file will not be protected.
IF\$UNP	Makes a protected file deletable. The file to be unprotected must not be open; otherwise, an error occurs.
IF\$REN	Directs RTDSP to replace the directory name of the file with the name specified in the alternate fields of the file identification block. The file to be renamed must not be open when this function is used; otherwise, an error occurs. Both file specifications must specify the same device, controller, and unit number. The file must not be open; otherwise, an error occurs.
IF\$DIR	Performs a lookup operation on the directory. Then the IF\$RDV function can be used to read a directory segment into the buffer. The size of the data buffer used to read directories must be exactly two blocks (1024 bytes). Appendix E describes the format of a directory.
IF\$DEL	Deletes the directory entry of the file specified. The file must not be open when this function is used; otherwise, an error occurs.
IF\$INI	Requests initialization of a directory on a device. An error occurs if you use this function for a device that cannot support a directory, such as a terminal.

## MICROPOWER/PASCAL FILE SYSTEM

### 5.2.3 Device Driver Process Replies

If a reply is requested, SQIO will wait for a reply from the device driver process. This reply uses the structure descriptor designated in the FB.REP field of the file identification block in use. By waiting for a reply, the process performing I/O can ensure that the I/O operation is complete before continuing. If no reply is requested, the application process may continue even though the device driver process has not finished the operation.

The IFSWAI function of SQIO will wait on the structure descriptor specified in the FB.REP field without performing an I/O operation. The FMSBSM function modifier can be used with IFSWAI to indicate to SQIO that the reply semaphore specified in the previous request was a binary semaphore.

### 5.2.4 File Status

The FB.STA field of a file identification block reflects the status of the open file. RTDSP uses six bits of the file status word. The other bits are reserved for use by DIGITAL-supplied Pascal I/O routines.

Code	Function
SRW	Specifies how other processes can access this file if SSHARE is set. If this bit is set, other processes can read and write with the file. If this bit is clear, other processes can only read this file.
SSHARE	Used to designate whether or not other processes can access this file.
SNEWFL	Set by SQIO when a new file is created.
SNFSIO	Set by SQIO to indicate that this file is on a file-structured medium without a directory. For instance, if the contents of a disk or tape are written as a single file, no directory is included on the medium.
SNESDV	Set by SQIO to indicate that the device is non-file-structured.
SEOF	Set by SQIO when the end of file is reached after a virtual read or write.

## 5.3 PARSING FILE SPECIFICATIONS

The SPARSE routine parses MicroPower/Pascal file specifications. SPARSE retrieves a file specification from a specified location in memory, parses it, and places the parsed file specification in another specified location. Parsed file specifications have the same form as a specification in a file identification block.

## MICROPOWER/PASCAL FILE SYSTEM

If a field of a file specification supplied to \$PARSE is omitted, the corresponding fields of the parsed file specification buffer remain unchanged. Therefore, the buffer into which the parsed specification will be placed should be initialized to spaces or default values before \$PARSE is called.

All alphabetic characters must be uppercase; otherwise, an error will occur.

### 5.3.1 File Specification Syntax

The syntax of a file specification is:

DEV:[UIC]filename.ext;version

Parameter	Definition
DEV:	Specifies a device name. This part of the file specification is the device specification; it contains up to three parts: <ul style="list-style-type: none"><li>• A device name, such as DY</li><li>• A controller name, such as A</li><li>• A unit number, such as 1</li></ul>
[UIC]	Specifies a user identification code for the file in the form [P,PN], where P and PN are integers between 0 and 255. This parameter is not included in directory entries and should not be used to distinguish between MicroPower/Pascal files.
FILENAME.EXT	Specifies the name and extension of the file. Although the file name can be up to nine characters long, only the first six characters are recorded in a directory entry. A difference in the last three characters does not produce a unique file name.  The extension can be up to three characters long.
	The legal character set for file names and extensions includes only uppercase letters and numbers.
version	An integer between 0 and 32767. This parameter is not retained in directory entries and should not be used to differentiate MicroPower/Pascal files.

### 5.3.2 File Specification Examples

#### 1. DYAA:TEST.DAT

Identifies the file 'TEST.DAT' on unit A of RXA2 controller A

## MICROPOWER/PASCAL FILE SYSTEM

### 2. DYA0:

Identifies drive 0 of RX02 controller A as a device without a directory.

### 5.4 PREDEFINED MACROS

The MicroPower/Pascal file system includes a group of predefined macro requests to simplify programming.

#### 5.4.1 FSINT\$

The FSINT\$ macro defines the symbols used by other file system macro requests. These symbols include the function calls to the \$QIO routines, the symbolic offsets into a file identification block, and the bits of the file system status word (FB.STA). You must use the FSINT\$ macro in the symbol definition section of your program.

The syntax of the FSINT\$ macro is:

FSINT\$

#### 5.4.2 QIOS

The QIOS macro pushes the address of the structure descriptor, the address of the file identification block, and the reply value on the stack. The function code is placed in the FB.FUN field of the specified file identification block. QIOS then calls the \$QIO subroutine.

**5.4.2.1 Macro Syntax** - The arguments to QIOS can be supplied in either positional or nonpositional format. The QIOS macro syntax is:

QIOS [sdb,] fib, func [,seq] [,reply]

Parameter	Definition
-----------	------------

sdb	The address of the structure descriptor ID for either the device driver process to receive the request for I/O or for RTDSP. If not specified, the FB.DVR field of the specified file identification block is used. This argument has the form: [sdb=] sdb-address
-----	---

fib	The address of the file identification block for the file. This argument has the form: [fib=] fib-address
-----	--

func	The function code for the operation being requested. This argument has the form: [func=] function-symbol
------	---

## MICROPOWER/PASCAL FILE SYSTEM

**seq** An optional sequence number that permits differentiation of otherwise identical requests. If specified, this number is placed in the FB.SEQ field of the specified file identification block. If no sequence number is specified, the value in FB.SEQ is incremented. This argument has the form:

[seq=] sequence-number

**reply** May be YES or NO. The default value is YES. YES specifies that \$QIO should wait for the reply from the device driver process before returning; NO specifies that \$QIO should not wait. A valid reply semaphore structure descriptor block must be supplied in the FB.REP field even if a reply is not requested. This argument has the form:

[reply=] reply-value

### 5.4.2.2 Syntax Examples - Examples of typical uses of the QIO\$ macro follow:

```
QIO$ sdb=#DSPSDB,fib=#INFIL,func=#IF$ENT,seq=#1,reply=YES
```

This request to \$QIO would, if possible, open a file. The device and the name of the file are specified in the file identification block, as are the attributes of the file.

```
QIO$ #DYA,#OUTFIL,#IFSWTV,,NO
```

This request to \$QIO would write out the data in the buffer pointed to by FB.BFP of the file identification block OUTFIL to a file on RX02 controller A. The unit number and file name are also specified in OUTFIL. The starting block of the file and the location of the device must have previously been filled in by opening the file. FB.BNO must contain the virtual block number to be written to.

```
QIO$ sdb=#DYA,fib=#INFIL,func=#IF$RDV+FMSINH,seq=#0,reply=NO
```

This request would read a buffer of data from the file identified by INFIL. The location of the file on the device and the size of the file must be known as a result of a previous call to \$QIO to open the file. The buffer location and the length of the file must also be in the file identification block. The function modifier FM\$INH has been added to this read request to inhibit retries by the device driver process.

### 5.4.3 FIB\$

The FIB\$ macro request creates a data area 110(octal) bytes long for use as a file identification block. File identification blocks contain information about open files. Most of these fields must be filled in before \$QIO is called. One file identification block must be created for each open file.

## MICROPOWER/PASCAL FILE SYSTEM

**5.4.3.1 File Identification Block Fields** - Figure 5-1 shows the fields of the file identification block. These fields are identified by symbolic offsets defined by the FSINT\$ macro request.

The alternate fields of the file identification block are used when performing rename operations, wherein two file names are required. In the alternate format, words 52 to 76(octal) contain the second file specification and can be filled by the \$PARSE subroutine.

The fields of file identification blocks are described below.

Field	Definition
FB.PTR	Reserved for use by DIGITAL-supplied Pascal I/O routines for the address of the buffer record to be accessed next.
FB.STA	A word of 1-bit status flags that describe the status of a file. Of the 16 bits in this word, 6 are required by the MACRO-11 programmer; the rest are reserved for use by DIGITAL-supplied Pascal I/O routines.
FB.DVR	The name of structure descriptor block of the queue semaphore with which \$QIO communicates with the device driver process or RTDSP. This field must be filled in before the call to \$QIO.
FB.CTL	A 1-character controller designation of the device on which the file resides -- for instance, the A in DYAO. This field can be filled by the \$PARSE subroutine.
FB.UNI	The binary device unit number -- for instance, the 0 in DYAO. This field can be filled by the \$PARSE subroutine.
FB.DVC	The 2-character name of the device on which the file resides -- for instance, the DY in DYAO. This field can be filled by the \$PARSE subroutine.
FB.UIC	Reserved for a user identification code (UIC), which the \$PARSE routine accepts, but which is neither retained in the directory nor used to differentiate open files. This field can be filled by the \$PARSE subroutine.
FB.FIL	The name of the file in ASCII characters. Although nine bytes are provided, RTDSP uses only the first six bytes. File names of less than nine characters must be padded on the right with spaces. This field can be filled by the \$PARSE subroutine.

## MICROPOWER/PASCAL FILE SYSTEM

0	Ptr to next record	FB.PTR
2	Status	FB.STA
4	Driver structure ID	FB.DVR
6	Structure ID (cont)	
10	Structure ID (cont)	
12	Unit   Controller	FB.UNI/FB.CTL
14	Device	FB.DVC
16	UIC	FB.UIC
20	File name	
22	File name (cont)	FB.FIL
24	File name (cont)	
26	File name (cont)	
30	Name(cont)   Extension	FB.EXT
32	Extension (cont)	
34	Version	FB.VER
36	Reply structure ID	
40	Reply ID (cont)	FB.REP
42	Reply ID (cont)	
44	Function	FB.FUN
46	Completion status	FB.CST
50	Sequence number	FB.SEQ
52	File size	FB.FSZ
54	Virtual block number	FB.VNO
56	Buffer pointer	FB.BPF
60	2nd buffer pointer	FB.BF2
62	Buffer size	FB.BFZ
64	Record size	FB.RSZ
66	End of buffer	FB.END
70	Records per buffer	FB.RPB
72	Starting block	FB.SBK
74	Starting block (cont)	
76	High block	FB.HBK
100	High block (cont)	
102	Last block	FB.LBK
104	Last block (cont)	
106	Status word	FB.FST
110	Active file list ptr	FB.ACT
		Alternate Fields

Figure 5-1 File Identification Block Fields

## MICROPOWER/PASCAL FILE SYSTEM

FB.EXT	The extension of the file specification. Extensions of less than three characters should be padded on the right by ASCII spaces. This field can be filled by the SPARSE subroutine.
FB.VER	Reserved for a version number. Although the parser accepts a version number and can place it in a file identification block, it is not retained in the directory entry. This field can be filled by the SPARSE subroutine.
FB.REP	A copy of the structure ID of the structure descriptor block of the reply semaphore to be used by the device driver process and RTDSP to reply to the SQIO routine. To fill this field, a structure descriptor block is created with the CRSTSS request, and the first three words are copied into this field.
FB.FUN	The function to be performed with the file by SQIO.
FB.CST	The completion status of the last I/O request. A nonzero value indicates an error, as does the setting of the carry bit in the PSW by SQIO. Chapter 4 of the <u>MicroPower/Pascal-RT Messages Manual</u> describes these error codes.
FBSEQ	An optional sequence number of the I/O request.
FB.FSZ (or FB.UN2/FB.CT2)	The size of a new file to be created or the size of an existing file returned by RTDSP.  FB.UN2 and FB.CT2 are used when renaming a file to hold the unit number and controller name of the device. This field can be filled by the SPARSE subroutine. Since the device does not change in a rename operation, this field could be copied from the FB.UNI/FB.CTL field.
FB.BNO (or FB.DV2)	Contains the virtual block number to be accessed by the next virtual read or write operation.  When renaming a file, FB.DV2 is used for the device specification of the new file specification. This field could be either copied from FB.DVC or filled by SPARSE.
FB.BFP (or FB.U12)	Contains the address of the data buffer for this file.  When renaming a file, FB.U12 holds the UIC of the new file specification. This field can be either filled by SPARSE or copied from FB.UIC.
FB.BF2 (or FB.FI2)	Reserved for use by DIGITAL-supplied Pascal I/O routines to contain the address of an optional second I/O buffer.  The FB.FI2 field is used when renaming a file to hold the first two characters of the new file name.

## MICROPOWER/PASCAL FILE SYSTEM

FB.BFZ	The size of the buffer, in bytes. This value need not be a multiple of 512 bytes (1 block), but device driver processes for file-structured devices usually read or write in blocks. When the IFSINI function of SQIO is used to initialize a directory, this field specifies the size of the directory, in segments.
	When renaming a file, this field contains the third and fourth characters of the new file name.
FB.RSZ	Reserved for use by DIGITAL-supplied Pascal I/O routines to contain the record size, in bytes.
	When renaming a file, this field contains the fifth and sixth characters of the new file name.
FB.END	Reserved for use by DIGITAL-supplied Pascal I/O routines to contain the address of the end of the data buffer.
	When renaming a file, this field contains the seventh and eighth characters of the new file name.
FB.RPB (or FB.EX2)	Reserved for use by DIGITAL-supplied Pascal I/O routines to hold the number of records per buffer.  The FB.EX2 field is used in a rename operation to hold the last character of the new file name and the first character of the new extension.
FB.SBK (or FB.EX2/FB.VE2)	The starting block of the file on the device. RTDSP places the logical starting block number of the file in this field when a file is entered or looked up on a device with a directory. For a device without a directory, SQIO sets this value to 0. The second word of this field always equals 0.  When performing a rename operation, the FB.EX2 field contains the second and third characters of the new extension, and the FB.VE2 field contains the new version number.
FB.HBK	The highest logical block number used by this file at a given point during program execution. The second word of this field always equals 0.
FB.LBK	The last logical block of the device that is allocated to the file. The second word of this field always equals 0.
FB.FST	The file system status word reserved for use by SQIO and other DIGITAL-supplied routines.
FB.ACT	Reserved for use by DIGITAL-supplied Pascal I/O routines for a pointer to the next active file.

## MICROPOWER/PASCAL FILE SYSTEM

**5.4.3.2 Macro Syntax** - The FIB\$ macro should be used in the impure-data declaration area. The syntax of the FIB\$ macro is:

Symbol: FIB\$

**Parameter**      **Definition**

Symbol              The legal MACRO-11 symbol that will identify the file identification block.

### 5.4.4 PARSE\$

The most convenient way to access the \$PARSE subroutine is to use the PARSE\$ macro. The PARSE\$ macro pushes the arguments on the stack and then calls \$PARSE.

**5.4.4.1 Macro Syntax** - Parameters to the PARSE\$ macro may be supplied in either positional or nonpositional fashion. The syntax of the PARSE\$ macro is:

PARSE\$ buff, len, dest

**Parameter**      **Definition**

buff              The address of the buffer containing the file specification to be parsed. This argument has the form:

[buff=] buffer-address

len              The length of the file specification, in bytes. This argument has the form:

[len=] specification-length

dest              The destination address of the parsed file specification, typically the FB.UNI offset into an FIB. This argument has the form:

[dest=] destination-address

**5.4.4.2 Syntax Examples** - The two examples below show the form of the PARSE\$ macro.

PARSE\$      buff=#LNUF, len=R0, dest=#OUTFIL+FB.UNI

PARSE\$      #LNUF, R1, #OUTFIL+FB.UNI

## CHAPTER 6

### CLOCK SERVICES

This chapter describes the clock service process and the services it provides to user processes. Section 6.1 presents a brief functional description and internal overview of the clock service process. Subsequent sections describe in detail, for both Pascal and MACRO-11 users, the service request interface.

#### 6.1 CLOCK SERVICE PROCESS

The CLOCK process is a privileged process that uses the system clock -- normally, a 60 Hz line-time clock -- to maintain system time and date. The process also provides basic timer and clock services to other processes in the system. The timer and clock perform the following functions:

1. Signal a Semaphore After a Given Time Interval (SSI)
2. Signal a Semaphore Periodically (SSP)
3. Cancel a Previous Timer Request (CTR) -- applies to SSI/SSP requests
4. Set Time of Day and Date (STD)
5. Get Time of Day and Date (GTD)

These basic functions are sufficient for implementing more complex timing functions. A timing function such as "delay process x seconds" can be implemented, for instance, by requesting that a semaphore be signaled after x seconds, followed by a wait operation on the same semaphore. (More detailed application suggestions are given later.) The Set and Get Time functions allow operator communication processes to set and display the time of day and date.

To request a timer/clock service, a process must send a service request message -- packet -- to the CLOCK process's request queue semaphore, named SCLOCK. (The interface is similar to that for I/O service requests.) The format of the request and the reply messages for each service are described in later sections. The CLOCK process assumes that requesting processes use the Pascal SEND/RECEIVE or MACRO-11 SEND\$/RCVDS mechanisms to send the request packet and receive the reply packet, if any. Privileged processes may use the alternative PUT\_PACKET/GET PACKET or SGLQS/WAIQ\$ mechanisms if the packet put is compatible with the send-level mechanisms; that is, the packet header must contain a valid control byte.

## CLOCK SERVICES

The CLOCK process manages the system clock, servicing all interrupts from it. The process maintains system time since power-up and updates the time of day and date on clock ticks. Date servicing includes date correction at the end of the day, month, and year. The number of clock ticks per second produced by the system clock may differ from the usual 50 or 60 but must, of course, be an integer value.

The time and date are kept in two different internal formats. The system time is kept in a double-precision counter that represents the number of clock ticks since system power-up. This form is important for processing timer requests and for time stamping, since it is a compact representation of system time. (The time interval specified in timer requests is expressed in clock ticks.) The maximum system time that can be stored, assuming a 60 Hz clock, is approximately two years, three and one-half months.

The time of day and date are kept in another form, implemented as a vector of elements containing ticks, seconds, minutes, hours, days, months, and years since 1980. The ticks element is incremented and compared with a table value representing the number of ticks per second. If the new ticks value exceeds the table value, the ticks value is reset, and the next vector element, seconds, is incremented and compared to seconds per minute. This increment-and-compare procedure is continued as long as successive vector elements overflow when incremented.

### 6.2 SERVICE REQUEST INTERFACE

The request messages for the several timer/clock services vary in size and content; they are described in Sections 6.2.2 to 6.2.6. All the information in these messages is sent by value; no data by reference is required in any clock request packet, unlike many device-handler requests. In MicroPower/Pascal, for example, you would use the following form of the Pascal SEND procedure to send the 14-byte message required for a timer request (either SSI or SSP) to the \$CLOCK request queue:

```
SEND ( VAL_DATA := <request record>, VAL_LENGTH := 14,  
      PRIORITY := <request priority>,      (*Optional Parameter*)  
      DESC := <$CLOCK semaphore descriptor> );
```

Or, you could use the following:

```
SEND ( VAL_DATA := <request record>, VAL_LENGTH := 14,  
      PRIORITY := <request priority>,      (*Optional Parameter*)  
      NAME := '$CLOCK' );
```

Note that the REF\_DATA and REF\_LENGTH parameters are omitted in the call.

In MACRO-11, you would use the following form of the SEND\$ primitive call to send the same request packet:

```
SEND$ area,sdb,priority,#14.,message-buffer,#0,#0
```

The final two zero-valued parameters -- for rlen and rbuf -- indicate "no data by reference."

The CLKDFS macro defines clock request packet symbols. Offset symbols for the packet are defined relative to the beginning of the packet, including the standard (6-byte) packet header.

## CLOCK SERVICES

Correspondingly, all reply messages sent by the CLOCK process are sent by value. Therefore, to receive a reply message, when applicable, through your reply queue semaphore, you would use the following form of the RECEIVE procedure. Here we assume a 20-byte reply as is returned by the Get Time and Date service:

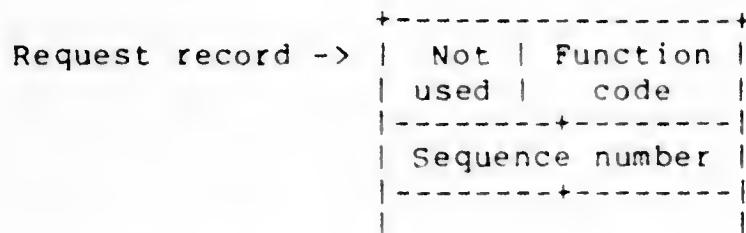
```
RECEIVE ( VAL_DATA := <reply record>, VAL_LENGTH := 20,  
          RET_INFO := <receive info. block>,  
          DESC := <reply semaphore descriptor> );
```

Or, you could use the following form of the RCVDS primitive call:

```
RCVDS area,sdb,receive-info-block,#20.,reply-buffer,#0,#0
```

### 6.2.1 Function Codes and Sequence Numbers

The first two words of the request message, the same for all service requests, contain a function code and a sequence number, as follows:



In the format above:

- Function code specifies the desired service function; its values are as follows:

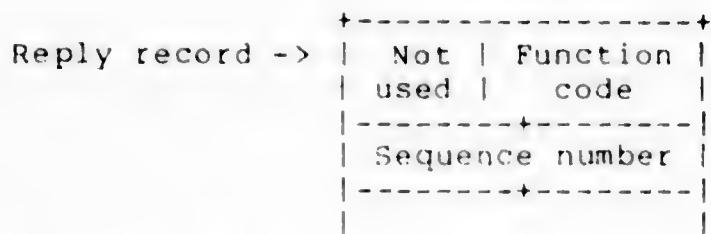
Value	Function
0 (CF\$SSI)	Signal After Interval
1 (CF\$SSP)	Signal Periodically
2 (CF\$CTR)	Cancel Timer Request
3 (CF\$STD)	Set Time and Date
4 (CF\$GTD)	Get Time and Date

The MACRO-11 function code symbols, in parentheses, are defined by the CLKDFS macro.

- Sequence number is an identifying number for a given request, supplied by the requestor. You can use this number in a cancel function to identify an individual unexpired timer request to be canceled.

Reply messages are returned for the Get Time and Date (CF\$GTD) function and for Cancel Timer Request (CF\$CTR) requests that cancel individual requests. The first two words of a reply message contain a function code and a sequence number, as follows:

## CLOCK SERVICES



In the format above:

- Function code specifies the service function for which a message was returned; its values are as follows:

Value	Function
7 (CF\$CTA)	Cancel Timer Request (for individual request)
8 (CF\$TAD)	Get Time and Date

- Sequence number is the requestor-supplied sequence number.

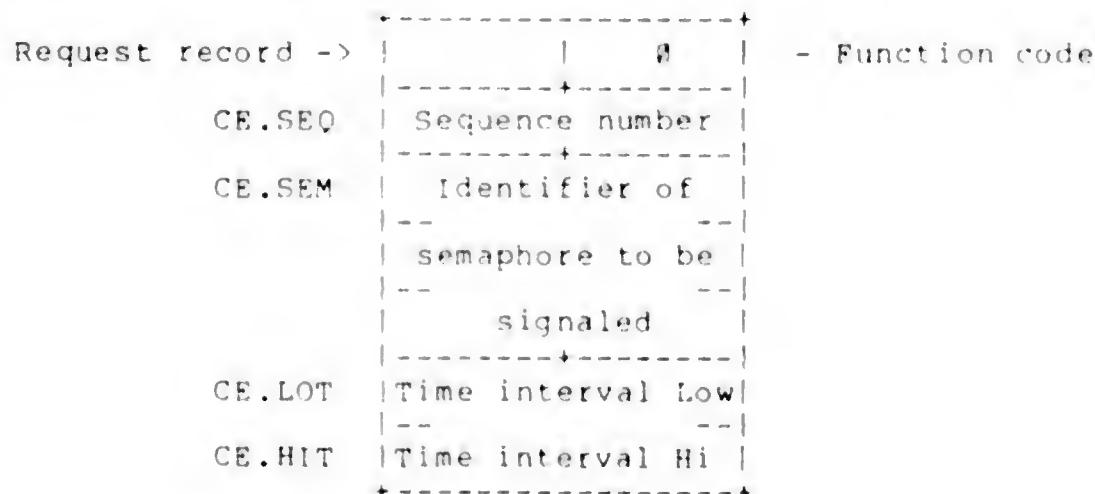
The individual timer and clock services and their corresponding request/reply messages are described in the following sections.

### 6.2.2 Signal a Semaphore After a Given Time Interval (SSI)

The SSI timer service signals the requestor's semaphore on expiration of a specified time interval, following receipt of the request. The time interval is expressed in the request as a number of clock ticks. Thus, if a 60 Hz clock is used, a time interval of 150 corresponds to a minimum delay of two and one-half seconds.

The semaphore to be signaled can be a binary, counting, or queue semaphore. For a binary or a counting semaphore, the CLOCK process deallocates the request packet after signaling. (There is no reply packet in this case.) The request sequence number must be set to 0 to indicate a binary or a counting semaphore. For a queue semaphore, the CLOCK process sends back the request message, unmodified, after the specified interval. The request sequence number must be nonzero to indicate a queue semaphore.

#### 6.2.2.1 SSI Request Message - The format of an SSI request message, excluding the standard packet header -- which is transparent to send/receive-level users, is as follows:



## CLOCK SERVICES

In the format above:

- CE.SEM is a Pascal STRUCTURE\_ID type variable -- first three words of the SDB in MACRO-II. If this field identifies a binary or a counting semaphore, the sequence-number value must be 0. If this field identifies a queue semaphore, the sequence-number value must be nonzero.
- CE.LOT and CE.HIT specify the time interval as a double-word unsigned integer value; CE.LOT contains the least significant part of the value, and CE.HIT contains the most significant part. The interval is expressed as a number of clock ticks.

Be careful that the sequence-number value and the type of semaphore identified in the request correspond as described above for field CE.SEM. If they do not -- for example, if a queue semaphore is identified and the sequence number is 0 -- a runtime error will occur in the CLOCK process, with possibly obscure results; the requestor's semaphore will never be signaled by CLOCK, and any process waiting on it may hang indefinitely.

**6.2.2.2 SSI Reply Message** - If the semaphore identified in the SSI request is a queue semaphore, the clock process will return the original request packet on that semaphore when the specified time interval expires. Thus, the request message and the reply message, if any, are identical for the SSI function.

**6.2.2.3 Applications** - An SSI request can be used to implement a variety of delay and time-out mechanisms. Some examples follow:

1. Delay for x seconds: A process can delay itself for x seconds by issuing a timer request to signal a semaphore after that period of time and by then doing a WAIT on that semaphore. The process will block on the semaphore until the CLOCK process signals it x seconds later. This type of delay can be used to wait for a lengthy asynchronous I/O operation to complete. It also could be used by a compute-bound process to free the processor for use by other, lower-priority processes.
2. Time out: A process can implement various time-out functions by using the SSI request. For example, a user-level device handler can start an I/O operation and then post a timer request for a time interval sufficient to complete the operation. If the I/O operation does not complete (interrupt) within the allotted time, the SSI service signals the time-out semaphore. The process waiting on the semaphore is activated and can take corrective action. If the I/O operation completes first, the handler can cancel the SSI request.
3. Wait until message is received or for x seconds: If a process is blocked while waiting for receipt of a message on a queue semaphore, the message may not arrive if there is a failure elsewhere in the application. The process is permanently blocked, and the system might deadlock. A way to avoid this situation is to request a timer signal on the message semaphore before doing the receive or wait-queue operation. If the message does not arrive within the specified timer interval, the CLOCK process will activate the

## CLOCK SERVICES

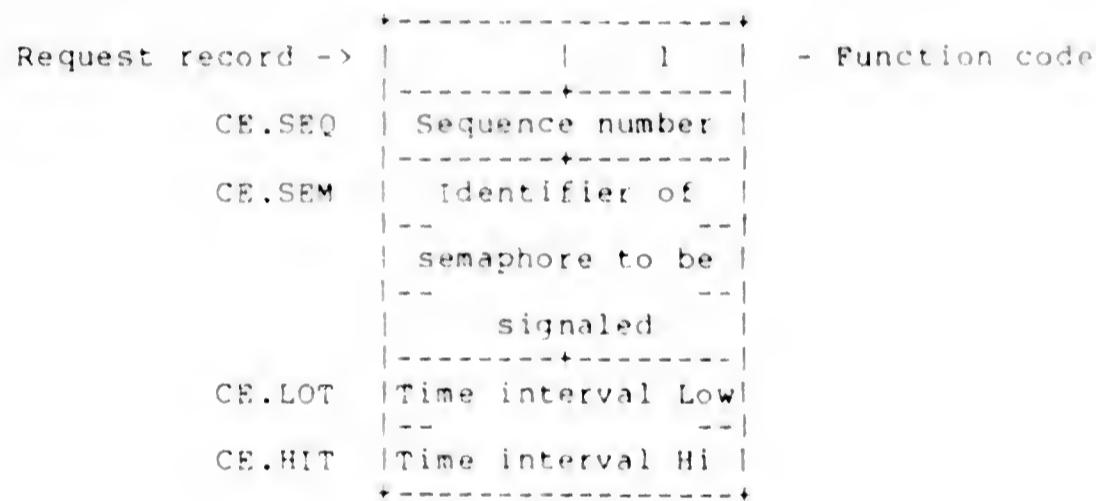
blocked process by sending its reply message to the message semaphore. The process must, of course, be able to distinguish between the desired message packet and the CCI reply packet. If the message does arrive, the timer request can be canceled.

### 6.2.3 Signal a Semaphore Periodically (SSP)

The SSP timer service signals the requestor's semaphore once every n clock ticks until the request is explicitly canceled. That is, the CLOCK process signals the semaphore n clock ticks after processing the request and repeats the signal every n ticks thereafter.

The semaphore to be signaled can be either a binary or a counting semaphore. (There is no provision for a reply message.)

**6.2.3.1 SSP Request Message** - The format of an SSP request message, excluding the standard packet header, which is transparent to send/receive-level users, is as follows:



In the format above:

- CE.SEM is a Pascal STRUCTURE\_ID type variable -- first three words of the SDB in MACRO-11. This field must identify a binary or a counting semaphore.
- CE.LOT and CE.HIT specify the time interval between signals as a double-word unsigned integer value; CE.LOT contains the least significant part of the value, and CE.HIT contains the most significant part. The interval is expressed as a number of clock ticks.

## CLOCK SERVICES

### 6.2.3.2 SSP Reply Message - None.

6.2.3.3 Applications - The SSP request produces a periodic signal function that has various applications. Several examples follow:

1. Periodic execution of a procedure: A procedure may be executed on a regular, periodic basis by using the SSP request. Two variations are possible by doing the wait either before or after the main body of the procedure. These variations affect only the initial characteristics of the periodic execution, not the steady state characteristics. If the procedure were part of a larger process loop, the procedure code could, as an alternative to blocking at the procedure, be conditionally entered or skipped, depending on the results of a conditional wait on the timing semaphore done before the procedure body.
2. Repetitive polling: A repetitive procedure could be used to poll an I/O device for status changes, such as checking the terminal keyboard interrupt enable bits or the modem control bits on communications devices lacking modem-change interrupts.

### 6.2.4 Cancel One or More Timer Requests (CTR)

The CTR service cancels one or more previously issued SSP or (unexpired) SSI requests to signal a given semaphore. If the sequence number in the CTR request is nonzero, the timer request queue is searched for a request with a matching semaphore identifier and sequence number. If more than one such request is found in the queue, the one with the earliest expiration time is canceled. If the sequence number in the CTR request is 0, all requests in the timer request queue with a matching semaphore identifier are canceled, without regard to their sequence numbers.

For a "cancel individual request" function, the CTR request message can contain a reply queue semaphore identifier; the CTR service returns a status reply message via this semaphore, indicating whether or not the specified timer request was found and canceled. No reply message is returned for a "cancel all requests" function; the reply queue semaphore field (CE.RSM) of the CTR request is ignored if the sequence number is 0.

6.2.4.1 CTR Request Message - The format of a CTR request message, excluding the standard packet header, which is transparent to send/receive-level users, is as follows:

## CLOCK SERVICES

Request record ->	+-----+     2    -----+-----+ CE.SEQ   Sequence number    -----+-----+ CE.SEM   Identifier of  -- timer request  --   semaphore  -----+-----+ CE.RSM   Identifier of  -- reply queue  --   semaphore +-----+	- Function code
-------------------	--	-----------------

In the format above:

- CE.SEQ, if nonzero, specifies the sequence number of the individual timer request to be canceled. If this value is 0, all timer requests having a matching CE.SEM will be canceled.
- CE.SEM is a Pascal STRUCTURE\_ID type variable -- the first three words of the SDB in MACRO-11. This field identifies the semaphore that is specified in the request(s) to be canceled.
- CE.RSM is a Pascal STRUCTURE\_ID type variable -- the first three words of the SDB in MACRO-11. This field identifies the queue semaphore through which a reply message is to be sent for an individual-request cancellation. (Its content is not significant if the CE.SEQ value is 0.) If the first word of this field is 0, no reply is sent in any case.

**6.2.4.2 CTR Reply Message** - For a "cancel individual request" function, the CLOCK process sends a 3-word reply, via the reply queue semaphore, on completion of the CTR function. The reply message contains a reply function code, the sequence number specified in the request, and a completion-status code, as follows:

Reply record ->	+-----+     7    -----+-----+ CE.SEQ   Sequence number    -----+-----+ CE.STS   Status code   +-----+	- Reply function code
-----------------	---	-----------------------

In the format above:

- CE.STS specifies the status of the completed CTR request; the status values returned are as follows:

Value	Function
0 (CE.RNF)	Specified timer request not found
1 (CE.OK)	Timer request canceled

The MACRO-11 status code values, in parentheses, are defined by the CLKDFS macro.

## CLOCK SERVICES

### 6.2.5 Set Time of Day and Date (STD)

The STD service allows a user process to set the time of day and date maintained by the CLOCK process. The time and date values to be set are specified in the request message. No reply message is provided by this service.

**6.2.5.1 STD Request Message** - The format of an STD request message, excluding the standard packet header, which is transparent to send/receive-level users, is as follows:

Request record ->	3	- Function code
CESEQ	Sequence number	
C1.MIN	Minute   Second	C1.SEC
C1.DAY	Day   Hour	C1.HRS
C1.YRS	Year   Month	C1.MON

In the format above:

- Second specifies the second of the current minute, ranging from 0 to 59(decimal)
- Minute specifies the minute of the current hour, ranging from 0 to 59(decimal)
- Hour specifies the hour of the current day, ranging from 0 to 23(decimal)
- Day specifies the day of the current month, ranging from 0 to last day minus 1(decimal)
- Month specifies the month of the current year, ranging from 0 to 11(decimal)
- Year specifies the number of years since 1980 in the current date (current year minus 1980)

Note that all request record value ranges begin at 0, including the ranges for day and month. This convention reflects the MicroPower/Pascal compiler's internal representation of ranges.

### 6.2.5.2 STD Reply Message - None.

## CLOCK SERVICES

### 6.2.6 Get Time of Day and Date (GTD)

The GTD service returns the current time of day and date to the requesting process in the reply message.

6.2.6.1 **GTD Request Message** - The format of a GTD request message, excluding the standard packet header, which is transparent to send/receive-level users, is as follows:

Request record ->	4	- Function code
CESEQ	Sequence number	
CESEM	Identifier of reply queue	
	semaphore	

In the format above:

- CE.SEM is a Pascal STRUCTURE\_ID type variable -- the first three words of the SDB in MACRO-11. This field must identify a queue semaphore through which the reply message will be sent.

6.2.6.2 **GTD Reply Message** - The GTD service sends a 20-byte reply message via the reply queue semaphore specified in the request. The reply contains a reply function code -- 8(decimal) or 10(octal) -- the sequence number specified in the request, the number of ticks since start-up, and current time-of-day and date values, as follows:

Reply record ->	8	- Reply function code
CESEQ	Sequence number	
CESEM	Ticks since start-up	
	Unused	
C2MIN	Minute   Second	C2SEC
C2DAY	Day   Hour	C2HRS
C2YRS	Year   Month	C2MON
CE.TIC	Ticks in second	
CE.TCS	Ticks per secnd	

## CLOCK SERVICES

In the format above:

- Second specifies the second of the current minute, ranging from 0 to 59(decimal)
- Minute specifies the minute of the current hour, ranging from 0 to 59(decimal)
- Hour specifies the hour of the current day, ranging from 0 to 23(decimal)
- Day specifies the day of the current month, ranging from 0 to last day minus 1(decimal)
- Month specifies the month of the current year, ranging from 0 to 11(decimal)
- Year specifies the number of years since 1980 in the current date (current year minus 1980)
- Ticks since start-up specifies the number of ticks modulo 232 since the clock process started
- Ticks in second specifies the number of clock ticks in the current second, ranging from 0 to ticks per second (CK\$TPS) minus 1
- Ticks per second specifies the number of ticks produced by the system clock per second (the clock frequency)

Note that all reply record value ranges begin at 0, including the ranges for day and month. This convention reflects the MicroPower/Pascal compiler's internal representation of ranges.

## CHAPTER 7

### EXCEPTION PROCESSING

Exceptions are events caused by hardware-detected errors, software-detected errors, and certain traps. The two general classes of exceptions are service traps and error traps. Service traps are explicit traps caused by executing trap instructions (TRAP, BPT, EMT). Service traps are usually intentional and can be used to request a service or intervention from a trap-processing routine. An error trap is usually not intentional and is caused by executing an instruction that fails -- illegal instruction, illegal address, memory-parity error, memory-management error, and so forth -- bus time-out errors, power failure/restart, and other hardware errors.

The MicroPower/Pascal kernel provides a mechanism for dispatching exception conditions to exception-handler processes or procedures (routines). When an exception occurs, it is vectored to the kernel, which dispatches the exception to an exception-handler process or procedure, if one exists. If no exception handler is provided, the kernel aborts the process causing the exception, and the process becomes inactive.

The remainder of this chapter discusses MicroPower/Pascal exception processing and especially exception conditions, types, and subcodes; kernel exception dispatching; and exception handlers. Before reading this chapter, you should carefully read the descriptions for the CCNDS, DEXCS\$, SERAS\$, and REXCS\$ kernel primitives in Chapter 3. If you plan to write exception-handler processes or procedures in Pascal, refer to the MicroPower/Pascal Language Guide, Chapter 17.

#### 7.1 EXCEPTION CONDITIONS, TYPES, AND SUBCODES

Exception conditions are identified by types, also called classes, and subcodes. There are 16 exception types; each exception condition is identified by an exception type symbol. Depending on the exception type, more than one exception may be identified by a type. Subcodes further identify the exception when multiple exceptions comprise an exception type. A complete list of exception types and subcodes and a brief description of each exception are given in Table 7-1.

MACRO-11 users can use the macro EXMSKS in COMU.SML or COMM.SML to define all exception types and subcodes symbolically. For Pascal users, all exception types and subcodes are declared in EXC.PAS.

## EXCEPTION PROCESSING

### NOTE

Do not confuse exception type and subcode, described above, with exception group code. Exception group code is a parameter specified when creating a process (see Sections 3.5 and 3.11). The group code parameter declares a process to be a member of an exception group. (An exception group is a set of processes grouped together because of common exception-handling requirements.) The group code can then be used -- in CCNDS or CONNECT EXCEPTION calls -- to establish one or more exception handlers for all processes in the exception group. See Sections 3.4 and 7.4 for more information on exception groups and the group code parameter.

Table 7-1  
Exception Types and Subcodes

Type	Subcode	Description
EX\$RPT	ESSNSC	Breakpoint trap (RPT = instruction executed, vector=14) No subcode set
EX\$EMT	0..255	Emulator trap (EMT = instruction executed, vector=130) User-specified subcode in the range 0-255
EX\$EXC	ESSBRK	Execution error Break (\$BC-11(2) only; vector=140)
EX\$HTO	ESSABT ESSATN ESSCTL ESSDAL ESSDRV ESSEOF ESSEVL ESSFOR ESSIBN ESSIDA ESSIFU ESSIVD ESSIVM ESSIVP ESSNXM ESSNXU ESSOFL ESSOVF ESSOVR ESSPAR ESSPNA	Hard I/O error Handler request aborted Device attention required Controller error Device already allocated Drive error End of file on device End of volume Format error Invalid block number Invalid device address Illegal I/O function code Invalid data Invalid mode Invalid parameter Nonexistent or read-only memory Nonexistent unit Device off line or not mounted Overflow Overrun Parity error Packet not available to perform request

(Continued on next page)

## EXCEPTION PROCESSING

Table 7-1 (Cont)  
Exception Types and Subcodes

Type	Subcode	Description
	ESSPWR	Power failure
	ESSPD	I/O processing stopped
	ES\$TIM	Device timeout
	ESSUNS	Unsafe volume
	ESSVOL	Invalid volume
	ESSWLK	Write-locked unit
EX\$IOP		Illegal operation
	ESSFOP	Floating-opcode error
	ESSILL	Illegal instruction (vector=010)
EX\$MEM		Memory fault
	ESSBUS	Bus error (timeout, vector=004)
	ESSMMU	Memory-management unit (MMU) error (LSI-11/23 only; vector=114)
	ESSMPT	Memory-parity error (LSI-11/23, only; vector=114)
	ESSVEC	Vector fetch (SBC-11/21 only; vector=000)
EX\$NUM		Numeric error
	ESSCON	Conversion
	ESSEOF	Exponential overflow
	ESSFDZ	Floating-point divide by 0
	ESSFOV	Floating-point overflow
	ESSFUN	Floating-point underflow
	ESSIDZ	Integer divide by 0
	ESSINM	Negative integer modulus
	ESSIOV	Integer overflow
	ESSLNP	Log of a nonpositive value
	ESSSRN	Square root of a negative value
	ESSUDV	Undefined variable
	ESSUDZ	Unsigned divide by 0
	ESSUOV	Unsigned overflow
EX\$RAN		Range error
	ESSASO	Array subscript out of bounds
	ESSCSO	Case selector out of range
	ESSNII	Reference of a NIL pointer
	ESSPCC	Program consistency check
	ESSSEO	Set element out of range
	ESSSTO	Stack overflow
	ESSSTU	Stack underflow
	ESSVSE	Variable subrange exceeded
EX\$RSC		Resource error
	ESSDDP	DISPOSE of already disposed pointer
	ESSFDF	File system DISPOSE failed
	ESSNLZ	NEW of length 0
	ESSNMB	Insufficient space for data buffer
	ESSNMF	Insufficient space for file variable
	ESSNMK	Insufficient space for kernel structure
	ESSNMP	Insufficient space for Pascal structure
	ESSNML	Insufficient space for stack

(Continued on next page)

## EXCEPTION PROCESSING

Table 7-1 (Cont)  
Exception Types and Subcodes

Type	Subcode	Description
EX\$RS1	(Reserved)	
EX\$RS2	(Reserved)	
EX\$SIO		Soft I/O error
	ESSBIV	Illegal Boolean value
	ESSCDA	Attempt to delete active file
	ESSCLS	Close error
	ESSCNT	Contention error
	ESSDAS	Direct access on sequential file
	ESSDCF	Device full
	ESSDIO	Directory I/O truncated
	ESSDRE	Directory read error
	ESSDRF	Directory full
	ESSDSF	Attempt to signal directory structure process (DSP) failed
	ESSDVF	Attempt to signal driver (device handler) failed
	ESSDWE	Directory write error
	ESSFIV	Illegal floating-point value
	ESSFNF	File not found
	ESSFNO	File not open
	FSSFNR	File not reset
	ESSFNW	File not rewritten
	ESSFRO	No write access allowed
	ESSICD	Invalid configuration data
	ESSIDR	Illegal directory
	ESSIDS	Illegal device specification
	ESSIFN	Illegal function
	ESSIFS	Illegal file specification
	ESSIFW	Illegal field width
	ESSIIV	Illegal integer value
	ESSILR	Illegal ring buffer name
	ESSILS	Illegal semaphore
	ESSIRS	Illegal rename specification
	ESSIUP	Illegal use of UPDATE parameter
	ESSLKP	Lookup error
	ESSPIT	Partial I/O transfer
	ESSPRO	Protection error
	ESSREF	Read past EOF
	ESSRSZ	Record size of 0 specified
	ESSUFN	Unsupported function
	ESSUIV	Illegal unsigned value
	ESSWEF	Write past EOF

(Continued on next page)

## EXCEPTION PROCESSING

Table 7-1 (Cont)  
Exception Types and Subcodes

Type	Subcode	Description
EX\$SVC		System service error
	ESSAOV	Already owned vector
	ESSCDN	Cannot specify both descriptor and name
	ESSEPN	Exception procedure not defined
	ESSIAD	Invalid address
	ESSIPM	Illegal parameter
	ESSIPR	Illegal process identifier
	ESSIQS	Illegal queue semaphore identifier
	ESSIRB	Illegal ring buffer identifier
	ESSISM	Illegal semaphore identifier
	ESSIST	Illegal structure identifier
	ESSIVC	Illegal vector
	ESSMDN	Must specify descriptor or name
	ESSMSP	Must specify process descriptor
	ESSMSS	Must specify structure descriptor
	ESSRDE	Reply descriptor expected
	ESSSIU	Structure is in use
	ESSSNI	Structure name already in use
	ESSUNV	Unowned vector
EX\$TRP	0.-255.	Trap (TRAP instruction executed, vector=034) User-specified subcode in the range 0-255
EX\$US1		(User-defined type and subcodes)
EX\$US2		(User-defined type and subcodes)

### 7.2 DECLARING EXCEPTIONS

A process can declare an exception -- also called a service trap -- by issuing a Report Exception (REXC\$) primitive. The REXC\$ primitive parameters specify the type of exception -- or error being simulated -- arguments to be passed to the exception handler, and number of bytes in the argument list. (See Chapter 3 for detailed syntax.) When an exception is declared in this manner, the kernel processes the exception in the same manner as for any other (non-REXC\$) exception.

### 7.3 EXCEPTION DISPATCHING

When an exception condition occurs, the trap is vectored to the MicroPower/Pascal kernel. The kernel first checks whether an exception-handler process is present for the group to which the process causing the exception belongs. If an exception-handler process does exist, the kernel places an exception stack frame on the stack of the process causing the exception, its context in its PCB, and places it in the exception-wait-active state. The PCB is then linked on the exception handler's queue. The kernel then declares a significant event; a scheduler pass is made before exiting the trap.

## EXCEPTION PROCESSING

At this point, the process causing the exception remains in the exception-wait-active state. The exception-handler process, when signaled by the kernel that an exception has occurred, returns to the ready-active state. The exception-handler process will then run on the basis of its process priority relative to those of other processes on the kernel's ready-active queue.

If an exception-handler process does not exist for the group of processes causing the exception, the kernel checks whether the process causing the exception has declared an exception routine for the exception type being processed. The kernel does this by examining the PCB of the process causing the exception. If an exception routine is contained within the process, an exception entry address will be present in PC.EXC. However, the kernel will not dispatch the exception to that address unless the exception is of a type that is included in the the PCB's exception type bit mask (PC.MSK). The exception type bit mask defines the exception type(s) specified in the Set Exception Routine Address (SERAS\$) primitive that established the exception routine.

If neither an exception process nor an exception routine is provided, the kernel immediately aborts the process by dispatching to the process termination entry point address (PC.TER) contained in the PCB. If you are debugging, PASDBG will then report the fatal (unhandled) exception.

### NOTE

Any attempt to continue program execution after the process has aborted may produce unpredictable results unless the application was specifically designed to allow the offending process to terminate without disruption of the application.

## EXCEPTION PROCESSING

Figure 7-1 shows the action of the kernel in processing an exception.

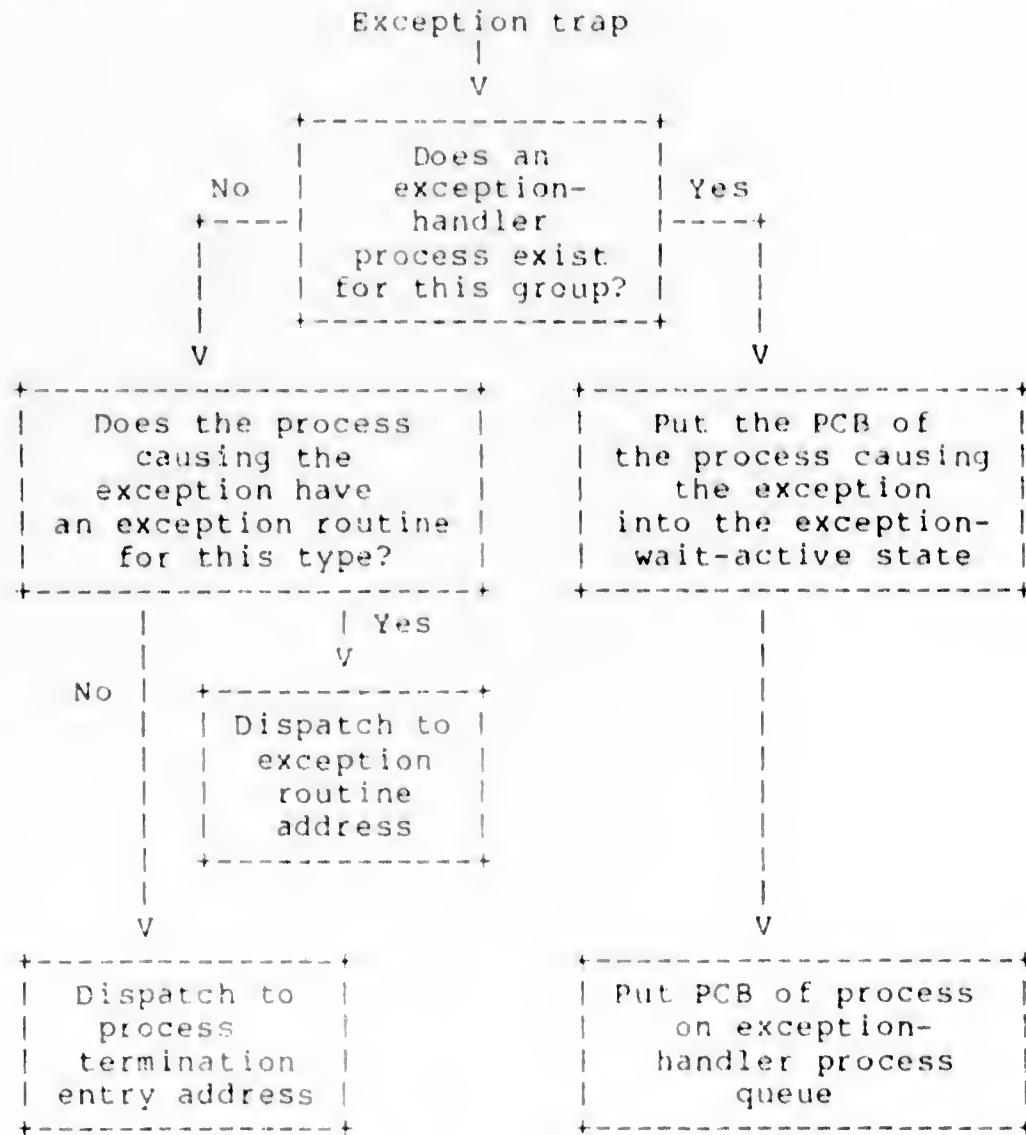


Figure 7-1 Kernel Exception Processing

## 7.4 EXCEPTION HANDLERS

### 7.4.1 Exception-Handler Processes

An exception-handler process is a privileged process that receives and manages exceptions of specified types for a group of user processes. Issuing a Connect to Exception Condition (CCNDS) primitive associates the exception group and type with the exception-handler process. This primitive specifies an existing queue semaphore on which exception conditions will be queued by the kernel. The exception-handler process can synchronize with the arrival of exceptions by using the Wait on Queue Semaphore (WAQS) primitive.

The group code parameter (grp) declares processes to be members of an exception group when they are created. The group code permits more than one exception handler for the same exception type to be present on the system, each implementing a different management strategy. An exception handler can also associate itself with all processes, regardless of group code. This is done by specifying a group code of  $\emptyset$  (wildcard value) for the grp parameter in the Connect to Exception Condition (CCNDS) primitive (see Section 3.4).

## EXCEPTION PROCESSING

Upon receiving an exception, the exception-handler process receives a pointer to the PCB of the process that caused the exception. That process has been blocked and placed in the exception-wait-active state by the kernel. The exception handler must determine a course of action, process the exception, and dispose of the PCB.

The exception handler is given each exception as it occurs. The exception-handler process must determine whether the process causing the exception wishes to receive the exception for processing by a declared exception routine and whether that action is desirable, based on the conditions causing the exception. It must also decide whether to service a request by itself.

Three courses of action are available to the exception handler, as follows:

1. Do some processing as required by the application and dismiss the exception, placing the process back on the ready queue or the ready-suspended queue. The process will resume executing at the instruction after the one causing the exception unless the saved PC was modified by the exception handler.
2. Pass the exception to the process, causing it to enter its exception entry point. This course of action should be taken if the PCB contains the exception entry point for the process (PC.EXC) and if the mask of exception types (PC.MSK) contains a bit that corresponds to the exception being processed.
3. Abort the process, causing it to enter its abort entry point. This action is appropriate if the exception represents a fatal situation that cannot be remedied by the exception handler or the process causing the exception.

The exception handler dismisses the exception by using the Dismiss Exception Condition (DEXCS) primitive. The three options above may be specified when requesting the dismissal action.

If an exception-handler process receives more than one exception type, it can create subprocesses to asynchronously manage single exception types or subtypes. The process priority for an exception-handler process should be higher than that of any process that may cause the exception.

### 7.4.2 Exception-Handler Routines

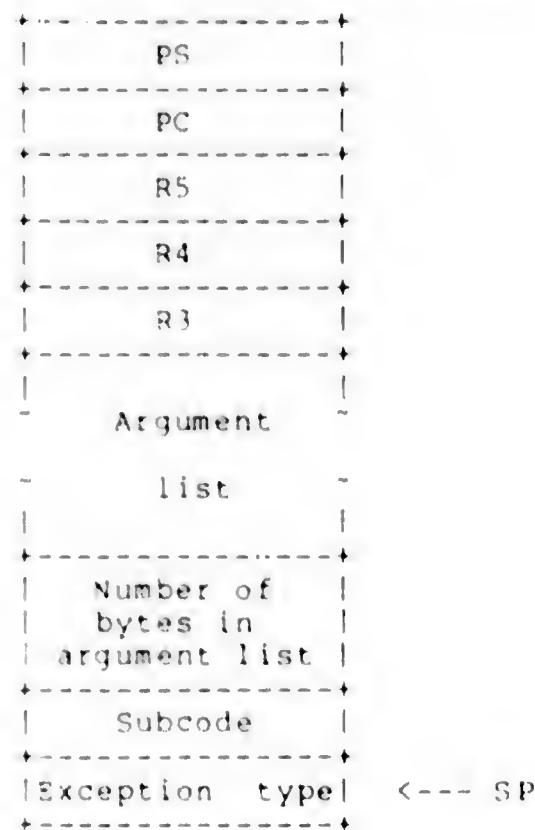
When an exception occurs, it is vectored to the kernel, which may dispatch it to an exception handler if one exists. If the exception handler chooses to allow it and/or if the process has requested that the particular exception be dispatched to it, the kernel dispatches it to the process's exception routine with a code indicating which type of trap occurred.

A process requests exception dispatching to an exception-handler routine by issuing the SERAS primitive, which supplies the exception routine entry address and bit mask defining the exception types it will service. This information is made available to the kernel via PC.EXC and PC.MSK parameters contained in the PCB.

## EXCEPTION PROCESSING

The exception routine is entered as though a software interrupt of the process had occurred, in full process context. Upon entry, the user's stack contains the PC and PS saved on servicing the exception trap, saved registers (R3, R4, R5), a list of arguments -- depending on the type of exception -- an argument count, exception subcode, and exception type (mask bit). General registers R3, R4, and R5 are available for use by the routine.

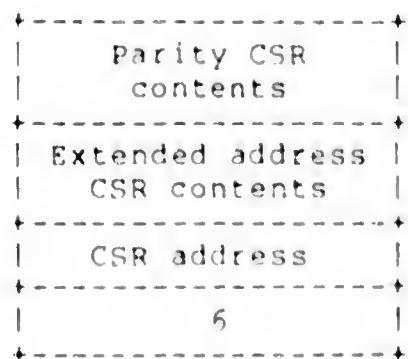
On entry to the exception service routine, the stack is as follows:



The argument list portion of the information the kernel places on the stack depends on the exception condition. Exceptions for which the kernel produces argument lists, the information contained in each argument list, and the number of bytes in each argument list are as follows:

### 1. Memory fault (EX\$MEM)

- Bus error (ESSBUS) -- No arguments
- Memory parity error (ESSMPT):



## EXCEPTION PROCESSING

- Memory-management unit error (ESSMMU)

MMU status	
register 0	
MMU status	
register 1	
MMU status	
register 2	
MMU status	
register 3	
10	

- Vector fetch (ESSVEC; SBC-11/21 only) -- No arguments

### 2. Illegal operation (EX\$IOOP)

- Illegal instruction (ESSILL) -- No arguments
- Floating opcode error (ESSFOP) -- No arguments

### 3. Breakpoint trap (EX\$BPT)

- No subcode used (ESSNSC) -- No arguments

### 4. Emulator trap (EX\$EMT)

- User-specified subcode -- No arguments

### 5. Trap (EX\$TRP)

- User-specified subcode -- No arguments

### 6. Numeric error (EX\$NUM)

- Floating-point processor exceptions, including conversion (ESSCON), unsigned overflow (ESSUDV), floating-point underflow (ESSFUN), floating-point overflow (ESSFOV), and floating-point divide by 0 (ESSFDZ):

Address of the	
instruction	
causing the	
exception	
2	

## EXCEPTION PROCESSING

### 7. Range error (EX\$RAN)

- Range errors, including stack underflow (ES\$STU) and stack overflow (ES\$STO):

Last known PC
value before
exception
condition
2

### 8. Execution error (EX\$EXC)

- Break (SBC-11/21 only) -- No arguments

After processing the exception, the routine must restore R3, R4, and R5 from the stack, purge the stack of the argument list, and exit by executing an RTI instruction.

## CHAPTER 8

### INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

This chapter discusses MicroPower/Pascal device handling, focusing on interrupt service routines (ISR) associated with device handlers. Before reading this chapter, you should read the descriptions for the CINT\$ and DINT\$ kernel primitives and the FORK\$ and P7SYSS kernel services in Chapter 3.

#### 8.1 DEVICE INTERRUPTS AND DEVICE-HANDLER FUNCTIONS

Interrupts are device hardware-generated requests that are sent to the processor hardware. Depending on hardware priority and the priority at which the processor is operating, the interrupt request will be serviced either immediately, if it is the highest-priority device requesting the processor, or later, following the current higher-priority operation. Interrupts serviced in this manner permit prioritized, event-driven processing involving I/O devices.

An interrupt is serviced by the processor after execution of the present instruction -- except floating-point instructions, which can be interrupted during execution. Once the instruction execution has been completed, the processor saves its present context (PC and PSW registers) before servicing the interrupt. After all interrupt processing -- including any other pending interrupt requests, fork processing, and interrupted kernel primitive processing -- has completed, the context of the interrupted program is restored, and processing continues as before.

I/O device handling in MicroPower/Pascal applications is done by device handlers -- software modules that contain processes, procedures, and routines. Included within the routines are the ISR and optional fork routine. The ISR is an essential part of the real-time environment of MicroPower/Pascal applications. The fork routine, an extension to the ISR, permits critical processing to be done by the ISR, followed by less critical processing in the fork routine. Fork routines can generally be interrupted by other interrupt requests. Thus, interrupt latency -- the time between a hardware-generated interrupt request and ISR entry -- is kept to a minimum. This is an important requirement in many real-time applications.

An interrupt results in entering an ISR only when an interrupt by the device it services is expected and desired. A kernel primitive (CINT\$) permits a device handler to connect the interrupt request to the ISR. Interrupt requests for a particular device result in normal service by the ISR only after the CINT\$ primitive has been issued. If

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

an ISR has not been connected to an interrupt vector or has been disconnected by the DINTS primitive, the interrupt is serviced by a kernel null ISR that dismisses the interrupt without performing any useful processing.

Entry to the handler's ISR or to the null ISR is made via an interrupt vector in low memory. Vectors for each hardware device are defined for the particular application by editing the configuration file. This file must be edited to reflect the device hardware configuration before building the handler software.

ISRs for device handlers written in Pascal must be written in MACRO-11. The CONNECT\_INTERRUPT procedure call translates the virtual addresses specified in the ISR source code into PAR 2 (code) and PAR 3 (data) values, using the contents of the user PARS.

ISRs for device handlers written in MACRO-11 can be written in either position-independent code (PIC) or non-PIC. With ISRs written in PIC, the CINT\$ primitive translates the virtual addresses as described for the device handler written in Pascal. However, when ISRs are not written in PIC, the addresses specified must be in PAR 2 or PAR 3 address space; otherwise, the CINT\$ request will fail, and an E.ADDR error will be returned.

### 8.2 THE INTERRUPT SERVICE ROUTINE

The ISR has the following functions:

- To respond quickly to interrupt requests issued by a hardware device with a minimum of context-switching overhead -- the time required to save part of the context of the interrupted process and enter the ISR
- To perform, at interrupt level, some limited time-dependent processing
- To signal, at fork level, a process that the event (interrupt) has occurred

In essence, the ISR processes hardware device events as they occur in real time. Less critical I/O processing is done by the device handler at process level.

The proportion of I/O processing performed by the ISR and the handler process, respectively, is determined largely by the characteristics of the device serviced and the nature of the I/O transfer. In general, deferring as much I/O processing as possible to process rather than to interrupt level (in the ISR) reduces interrupt latency for lower-priority devices.

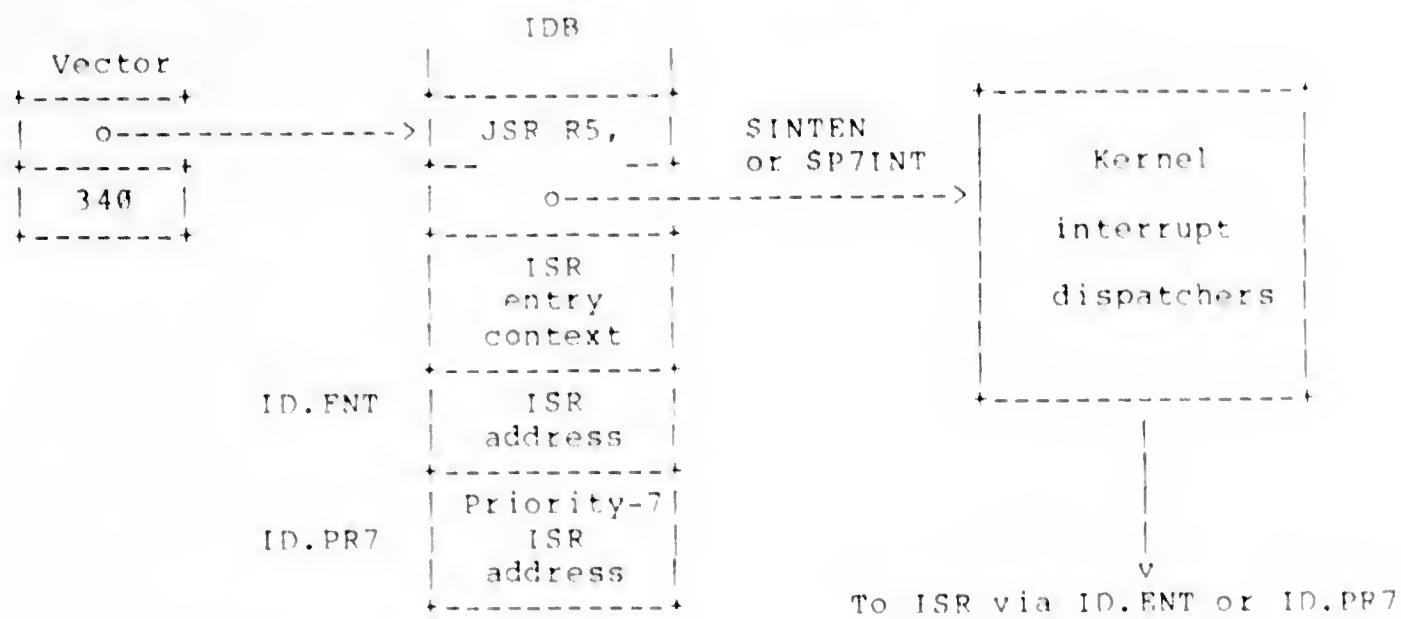
For example, a direct-memory-access (DMA) device, such as a disk controller, transfers large blocks of data without processor intervention, interrupting only when done with the transfer. As a result, the only interrupt-level processing that needs to be done is to awaken the handler process. The ISR issues a FORKS request -- enters fork-level processing -- and signals the handler process, indicating that an interrupt has occurred, that the requested I/O transfer has been completed, and that I/O processing at the process level can continue.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

In contrast, a device that transmits only one byte per interrupt at a relatively high and fixed transfer rate may require that almost all its I/O processing be done at interrupt level. This requirement is based on the excessive context-switching time that would be required by the frequent and numerous interrupts that would occur if data were processed by the device handler at process level while interrupts continued to occur. Thus, once a read or write operation is initiated by the handler process, the ISR completes a block-mode transfer to or from a device and a buffer and signals, at fork level, the handler process that the transfer was either successful or that an error was detected. In the case of an error, the process-level I/O processing by the device handler generally is responsible for determining whether a retry is required and, if so, for initiating the retry operation.

### 8.3 ISRs AND INTERRUPT DISPATCHING

The MicroPower/Pascal kernel receives device interrupts and passes them to appropriate ISRs through a 2-level dispatching mechanism. The transfer of CPU control on occurrence of an interrupt via the interrupt dispatch block (IDB) and the kernel interrupt dispatcher is as follows:



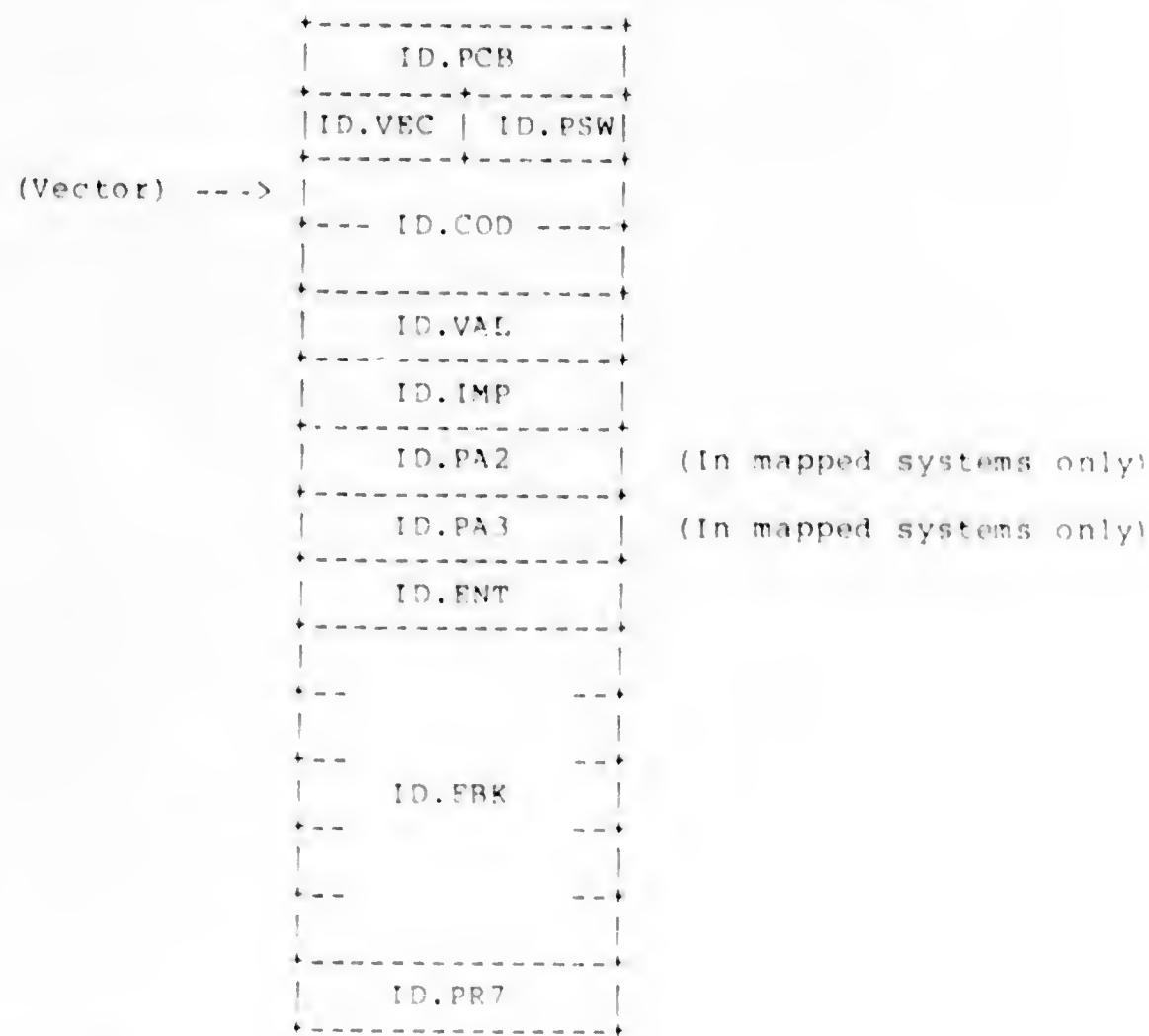
#### 8.3.1 The Interrupt Dispatch Block (IDB)

All interrupts are indirectly vectored to the kernel's interrupt dispatcher through an intervening data structure called the interrupt dispatch block (IDB). One IDB exists for each interrupt vector configured in a given system. Each IDB contains all the information needed for dispatching the interrupt that it uniquely represents, as well as an instruction that transfers control to the kernel's interrupt entry point.

A sufficient number of IDBs are statically allocated from the kernel's RAM data segment during the system-build process. Information from the DEVICES macro in the system configuration file determines the number of IDBs to be created.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

The format of an interrupt dispatch block is as follows:



In the format above:

- ID.PCB is the pointer to the PCB of the process owning the vector, if any -- that is, the process that performed the connect-interrupt operation.
- ID.PSW is the value of PSW priority and condition code (CC) bits for entry to ISR.
- ID.VEC is the scaled address of the vector associated with this IDB (address/2).
- ID.COD is the JSR R5, #SINTEN (or JSR R5, #SPINTINT) instruction (two words).
- ID.VAL is the value to be passed in R4 on normal ISR entry; specified in the CINTS call.
- ID.IMP is the pointer to the impure area; specified in the CINTS call.
- ID.PA2 is the kernel-mode PAR 2 value for mapping the ISR code segment. This IDB field exists only in a mapped environment.
- ID.PA3 is the kernel-mode PAR 3 value for mapping the ISR data segment. This IDB field exists only in a mapped environment.
- ID.ENT is the address of normal ISR entry point (that is, with priority less than 7).

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

- ID.FBK is the start of the fork block for this interrupt (five words).
- ID.PR7 is the address of priority-7 ISR entry point.

### 8.3.2 Kernel Interrupt Dispatcher

An interrupt causes processor control to pass to ID.COP in the vector's IDB. The IDB executes a JSR R5,SINTEN in the kernel, with R5 pointing to ID.VAL in the IDB. The functions performed by SINTEN are described below.

The interrupt dispatcher has two entry points: a normal entry point (SINTEN) for dispatching to ISRs that execute at a CPU priority less than 7 and a special entry point (SP7INT) for dispatching to ISRs that initially execute at CPU priority 7. (In both cases, the dispatcher is entered with interrupts inhibited at CPU priority 7.) The correct entry point for a given interrupt is set in the corresponding IDB.

When the interrupt dispatcher is entered for a normal ISR, it saves the full ISR register context (R3, R4, R5, PAR 2, and PAR 3), updates the kernel-state indicators to indicate the current level of processing, and establishes the ISR's entry context, using information stored in the IDB. (In a mapped system, the interrupt dispatcher also modifies the kernel-mode mapping.) It then dispatches to the ISR at the appropriate CPU priority, also indicated in the IDB.

When the interrupt dispatcher is entered for a priority-7 ISR, it bypasses its normal entry procedure -- except mapping the ISR code and data segment in a mapped system -- and immediately dispatches to the ISR. Priority-7 interrupts reduce interrupt-processing overhead significantly but impose severe restrictions on the ISR execution environment. During priority-7 execution, no other interrupts can be serviced until ISR execution is completed; thus, great care must be taken to keep priority-7 ISRs as brief as possible.

Thus, the interrupt dispatcher performs the following functions:

1. Saves, when dispatching to normal ISRs, general registers R3, R4, and kernel-mode mapping registers PAR2 and PAR3 in mapped systems on the stack; R5 is saved by the ISR R5 instruction in the IDB.

Saves, when dispatching to priority-7 interrupts, general registers R4 and R5, as well as PAR2 and PAR3 in mapped systems; R5 is saved by the ISR R5 instruction in the IDB.
2. Increments the interrupt nesting level. The interrupt level is initialized to -1 by INIT, which represents process level (no interrupts being serviced). The first interrupt raises the nest level to 0, marking the transition from process to interrupt level.
3. Switches, in an unmapped system, to the system interrupt stack if the nesting level is incremented to 0. In both mapped and unmapped systems, if the nesting level is 1 and if kernel primitive execution was interrupted, the stack is switched from the process kernel stack to the system interrupt stack.
4. Establishes, in a mapped system, the mapping context of the ISR.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

5. Establishes the priority and condition code bits in the PSW for the ISR by moving the ID.PSW field of the IDB to the PSW.
6. Dispatches, for normal ISRs, to the ISR, with R3 pointing to the impure area, R4 containing the value passed in the CINTS primitive, and R5 containing a pointer to the fork block for this interrupt. R3, R4, and R5 are available for use without explicit saving and restoring. All other registers must be saved before use and restored afterwards.

Dispatches, for priority-7 ISRs, to the ISR, with R5 containing a pointer to the ID.VAL field in the IDB and R4 containing the return address for the ISR; no other registers are saved, except PAR 2 and PAR 3 in mapped systems. R5 is available for use without explicit saving and restoring, provided that the ISR is not going to issue a P7SYSS followed by a FORKS. All other registers must be saved before use and restored afterwards.

### 8.3.3 Establishing the Interrupt-to-ISR Interface

An interrupt vector is connected to an ISR through the IDB in several distinct stages, beginning during system building. (Although much of the process is automatic and transparent to the user, understanding the process will aid your overall understanding of MicroPower/Pascal interrupt handling and the function of the CINTS primitive.)

**8.3.3.1 Allocating IDBs and Setting Vectors** - Every interrupt vector that will be used in an application must be declared in the system-configuration file so that a corresponding number of IDBs can be allocated during application building. The vectors are declared in the DEVICES macro. Each declared vector points to a unique IDB when the kernel memory image is constructed. (In ROM/RAM-based systems, it is assumed that the vector region will be located in ROM, since it is contiguous with the kernel code segment and part of kernel mapping. The power-fail/restart vector must also be located in ROM to permit system start-up on application of power.)

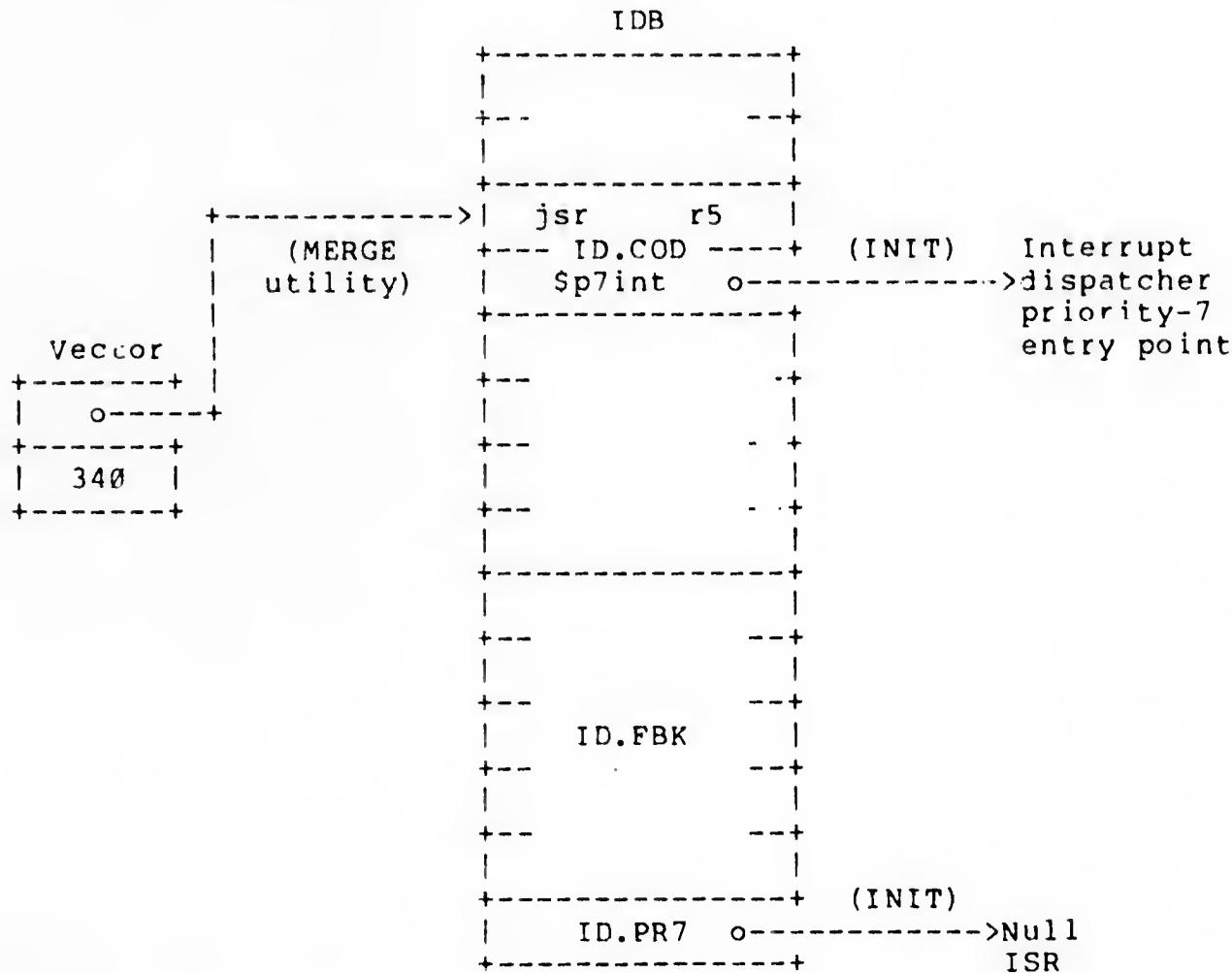
All vectors not declared as part of the hardware configuration point to a single null IDB; any undeclared interrupts will dispatch to the null ISR. The null ISR, which is located in the kernel, runs at priority 7. It increments a counter and returns -- executes an RTS R4 instruction -- dismissing the interrupt.

Since IDBs are dynamically modified by the CINTS kernel service, IDBs are always located in RAM.

**8.3.3.2 Initializing IDBs During Start-Up** - IDBs are statically allocated in the kernel's impure-data segment but are not statically initialized; that is, IDB contents are undefined. The system-initialization routine, INIT, initializes all IDBs during the start-up/restart sequence by directing each IDB to the null ISR.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

INIT also inserts a JSR R5 at \$P7INT instruction in IDB field ID.COD, which passes control to the interrupt dispatcher's priority-7 entry point when an interrupt occurs. INIT then places the null ISR entry address in IDB field ID.PR7, which is used for dispatching priority-7 ISRs. Thus, after system start-up, any unsolicited interrupts from declared vectors -- as well as unexpected interrupts from undeclared vectors -- are ignored until a proper connection has been made between a particular interrupt vector and an ISR, via the CINT\$ primitive. The linkages in place after system initialization are as follows:



**8.3.3.3 Connecting Interrupts to ISRs** - Each IDB remains in the state shown above, directed to the null ISR, until a device-handling process executes a CINT\$ primitive. (Processes written in Pascal use either the CONNECT SEMAPHORE or the CONNECT INTERRUPT procedure call to access the CINT\$ primitive.) This primitive associates a specified vector with the ISR specified in the CINT\$ call or implied in the case of CONNECT\_SEMAPHORE, by modifying the IDB assigned to the vector.

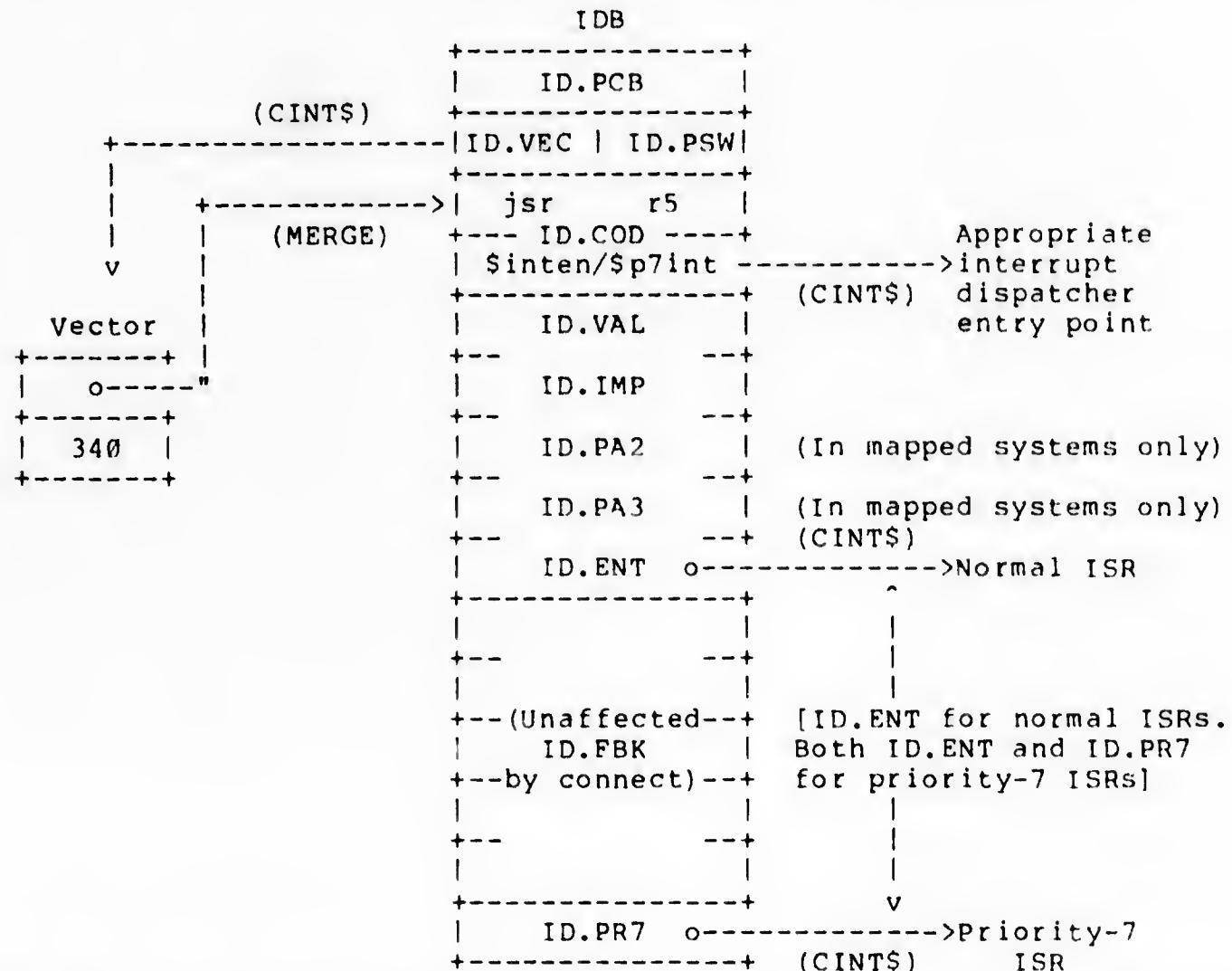
Specifically, executing the CINT\$ primitive does the following:

1. Places the appropriate dispatcher entry point address (\$INTEN or \$P7INT) in subfield ID.COD+2, depending on the priority level specified for the ISR in the CINT\$ call.
2. Places the ISR entry point address either in field ID.ENT, for a normal dispatch, or in field ID.PR7, for a priority-7 dispatch.
3. Links the IDB back to its vector via the ID.VEC field.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

4. Sets all other fields of the IDB, except ID.FBK, to the values specified or implied in the CINT\$ call; these fields contain ISR context-related information. (The fork-block substructure is not involved in the operation.)

The result of connecting the interrupt to the ISR is as follows:



Note that the process connecting a given interrupt vector becomes the owner of the vector/IDB involved; the owner's PCB index is placed in field ID.PCB of the IDB. Any subsequent CINT\$ operation specifying the same vector will fail and will return the busy/error code (E.BUSY). However, the owning process can disconnect the vector with the DINT\$ primitive, which reinitializes the corresponding IDB. The vector can then be connected to another ISR.

### 8.4 ENTERING AND EXECUTING ISRs

#### 8.4.1 Entering and Executing Normal ISRs

On entry to the normal ISR, the kernel interrupt stack is available for use by the ISR, and registers R3, R4, and R5 contain information that may be used by the ISR. R3 points to the ISR's impure area. R4 contains the val parameter specified in the CINT\$ call. (This parameter can be used to pass a device address or table index to an ISR.) R5 points to the fork block in the IDB; this pointer must be present when a FORK\$ call is issued by the ISR. The ISR runs at the priority specified by the ps parameter in the CINT\$ call, with the specified PSW condition code bits set on entry.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

Kernel primitives, except the FORKS kernel service, cannot be called from normal ISRs. In order to prevent reentrancy problems, the ISRs issue a FORKS call before issuing kernel primitives. This guarantees that an ISR issuing a primitive will not interrupt a process-level primitive operation, causing the kernel to be reentered. However, interrupts can be nested; that is, higher-priority interrupts can be serviced without causing reentrancy problems in the kernel.

### NOTE

Within a normal ISR, only R3, R4, and R5 can be used without first saving their contents. However, those registers may contain information needed by the ISR and, as a result, require saving. Within a priority-7 ISR, only R5 can be used without first saving its contents. However, the contents of R5 must be saved if the ISR issues a P7SYSS call.

### 8.4.2 Entering and Executing Priority-7 ISRs

A priority-7 specification in the CINT\$ call indicates a special case. Since running at priority 7 excludes all other interrupts, the ISR is dispatched without executing the normal interrupt entry procedure. High-priority interrupts can use this mechanism to perform critical operations requiring low interrupt latency. However, the ISR is responsible for saving and restoring any general registers it uses.

When entering a priority-7 ISR, R4 contains the return address, and R5 contains a pointer to ID.VAL in the IDB. The ISR can access its impure area by adding an offset to the address in R5 and point to ID.IMP; ID.IMP is a pointer to the impure area. For example, if ID.VAL contained the address of a device register having a word to be input to the first word of the impure area, the following MOV instruction would perform the function by using R5 addressing:

```
MOV @R5,@2(R5) ; Move word from device to buffer
```

Only the P7SYSS kernel service can be called from priority-7 ISRs. Thus, in order to issue a fork request, the ISR must first issue a P7SYSS call. The P7SYSS call sets the ISR priority to a specified value (less than 7) and establishes the ISR general register context, as on entry to a normal ISR. In other words, the primitive changes the context of a priority-7 ISR to a normal ISR context.

### 8.4.3 The Fork Routine

An ISR must issue a fork request before issuing any other kernel primitive. The fork request ends execution of the ISR at interrupt level, and the remainder of the ISR is executed as a deferred fork routine at interrupt priority 0. The fork request is made by issuing a \$FORK call, with R5 pointing to the fork block, as it was on entry to the ISR. The stack must be purged of any data pushed on it by the ISR before issuing the FORKS call. (A detailed description of the FORKS primitive is provided in Chapter 3.)

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

The kernel responds to the FORKS call by queueing the request on the kernel's fork request queue. The context of the ISR (R3, R4, and PC) is preserved in the fork block, along with the fork-level scheduling priority specified in the FORKS call.

The format of a fork block is as follows:

Pointer to -----> fork block	+-----+   FB.ADR   +-----+   FB.R3   +-----+   FB.R4   +-----+   FB.LNK   +-----+   FB.PRI   +-----+
------------------------------------	--

In the format above:

- FB.ADR is the PC of the requesting ISR -- the address to return to at fork-processing level.
- FB.R3 is the saved R3 of the requesting ISR.
- FB.R4 is the saved R4 of the requesting ISR.
- FB.LNK is the link word for the fork request queue.
- FB.PRI is the fork-level scheduling priority requested for this fork routine (0 to 65535); the fork request queue is ordered according to this value.

Fork routines must not issue kernel primitives that may cause them to block while waiting for an event. Fork routines may perform signal operations on semaphores but may not wait on a semaphore, which also could block operation if no signal were pending. Conditional waits may, however, be used, since they never block.

### 8.4.4 Dismissing the Interrupt

The normal ISR dismisses the interrupt by issuing an RTS PC instruction. However, before dismissing the interrupt, the routine must first clean the stack and restore any registers that were saved by the ISR on ISR entry. The interrupt can be dismissed in this manner when in interrupt level or when executing a fork routine.

The priority-7 ISR dismisses the interrupt by issuing an RTS R4 instruction. However, before dismissing the interrupt, the routine must first clean the stack and restore any registers that were saved by the ISR on ISR entry.

## INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES

### 8.5 KERNEL INTERRUPT EXIT PROCESSING

A return from an ISR is always made via either an RTS PC -- for normal interrupts -- or an RTS R4 -- for priority-7 interrupts -- instruction execution. (If a priority-7 ISR issues the P7SYSS call, it becomes a normal ISR, exiting via the RTS PC instruction rather than the RTS R4 instruction.) Processing in the kernel on return from an ISR depends on whether a lower-priority ISR was previously interrupted, kernel primitive execution was interrupted, a fork routine was interrupted, or a process was interrupted. These four ISR return conditions are listed below:

1. If returning to an interrupted ISR, the kernel decrements the interrupt nesting level and restores R3, R4, and R5, as well as kernel mapping registers PAR2 and PAR3 in mapped systems.
2. If returning to kernel primitive execution, the kernel switches to the per process kernel stack in use by the primitive, decrements the interrupt nest level, and restores registers R3, R4, and R5, as well as kernel mapping registers PAR2 and PAR3 in mapped systems.
3. If returning to a fork routine, the kernel decrements the interrupt nesting level and resumes the interrupted fork routine execution, regardless of fork routine priority. If one or more additional fork routines are queued, the interrupted fork routine is executed first, followed immediately by the remaining fork routines, based on their priority on the fork queue. No return is made to process-level execution until all ISR and fork routine execution has completed.
4. If returning to process-level execution, the kernel decrements the interrupt nesting level, restores registers R3, R4, and R5, as well as kernel mapping registers PAR2 and PAR3 in mapped systems, processes any significant events by calling the scheduler, and, in unmapped systems, switches to the user's stack and resumes process-level execution.

### 8.6 PASCAL LANGUAGE ISR INTERFACE

Device handlers can be written in Pascal for all processing except the ISR and fork routine; those routines must be written in MACRO-11. However, the MicroPower/Pascal language extensions allow handlers written in Pascal to associate interrupts with their ISRs and to signal semaphores when interrupts occur. These extensions provide the equivalent function of the CINTS primitive issued by handlers written in MACRO-11. The predeclared procedures (language extensions) are listed below:

Predeclared Procedure	Function
CONNECT_SEMAPHORE	Associates an interrupt vector with a semaphore so that the semaphore is signaled each time an interrupt occurs.
DISCONNECT_SEMAPHORE	Breaks the connection between an interrupt vector and the semaphore to which it was connected so that interrupts from the vector are ignored.

## **INTERRUPT DISPATCHING AND INTERRUPT SERVICE ROUTINES**

**CONNECT\_INTERRUPT**

Associates an interrupt vector with an interrupt service routine in order to establish a process as a device handler.

**DISCONNECT\_INTERRUPT**

Breaks the connection between an interrupt vector and an interrupt service routine so that interrupts from the vector are ignored.

Detailed descriptions of these predeclared procedures are provided in Chapter 16 of the MicroPower/Pascal Language Guide.

## CHAPTER 9

### GUIDE TO WRITING A DEVICE HANDLER

#### 9.1 DEVICE-HANDLER OVERVIEW

This chapter explains how to write custom device handlers -- device handlers 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 handlers and the kernel. However, because of the intimate relationship between the kernel and device handlers, some unavoidable changes may occur as new features are added to new versions of the MicroPower/Pascal product. Thus, device handlers written for this first version of MicroPower/Pascal may require modification for use with later versions.

A device handler, also called a device driver, is a set of processes, routines, and tables that process I/O requests for a particular hardware device or device controller. In general, device handlers are restricted to device-specific aspects of I/O processing; device-independent I/O processing common to other handlers or system services is performed by other system components.

Device handlers process I/O requests by performing many or all of the following functions:

1. Defining the peripheral device for the system
2. Preparing the device hardware and/or its controller for operation at system start-up and during recovery from a power failure
3. Performing any necessary device-dependent I/O preprocessing
4. Translating programmed requests for I/O operations into device-specific hardware commands
5. Activating -- starting or enabling -- the device
6. Responding to any interrupt requests generated by the device hardware; a device handler can also poll devices
7. Responding to requests to stop (abort) the I/O operation
8. Reporting device errors to the error-recording process
9. Returning completion status -- successful completion or error -- to the process that requested the I/O operation

## GUIDE TO WRITING A DEVICE HANDLER

A process requests I/O service from a particular device by sending a request packet to the device handler's request queue. The communication between the requesting process and the device handler, including request and reply packet formats, is described in detail in Section 4.1. Interrupt dispatching and interrupt service routines are described in detail in Chapter 8. You must become thoroughly familiar with the technical material presented in those sections before attempting to write a device handler.

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 handler.) 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 handlers 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 handler to your application, you will generally write or edit three separate source modules:

1. The handler prefix module (xxPFX.MAC or xxPFX.PAS)
2. The handler impure-area definition macro (xxISZ\$ in DRVDEF.MAC) or program (if Pascal)
3. The handler proper (xxDRV.MAC)

Device handlers designed and written by DIGITAL include the three source modules listed above. The handler prefix module and impure-area definition macro provided for standard device handlers (supplied by DIGITAL) make it easier for you to modify handler operations to conform to your hardware configuration. However, if the configuration of your hardware is not likely to change, you can write a device handler without using either the handler prefix module or the handler impure-area macro.

When designing and writing device handlers, carefully consider the conventions presented in the remaining sections of this chapter. DIGITAL discourages the use of kernel interfaces in device-handler 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 refer to handler source files -- such as DYDRV.MAC, the RX02 device handler -- and other referenced source modules supplied in your MicroPower/Pascal distribution kit as a guide for designing and writing your custom device handler.

The remaining sections in this chapter contain specific guidelines for writing each module.

## GUIDE TO WRITING A DEVICE HANDLER

### 9.2 DEVICE-HANDLER PREFIX MODULE

The device-handler prefix module statically allocates the impure area required for the device handler and defines certain device-specific parameters, such as:

- Priority values for the handler-initialization process, fork routine, request-handling process, and device hardware interrupt priority
- Number of hardware device controllers that the handler 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 made available to the handler in the form of tables of constants and text.

One device-handler prefix module is generally required for each device handler. The module name has the format `xxPFX.MAC` (or `xxPFX.PAS`), where `xx` identifies the device handler supported by the module. For example, `DYPFX.MAC` is the device-handler prefix module for `DYDRV.MAC`, the `RX02` device handler.

A handler 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 handler. The specific macros included in the prefix module are the handler configuration macro (`DRVCFS`) and the controller configuration macro (`CTRCFS`). (Because of its special requirements, the serial line (`XL`) device handler invokes neither the `DRVCFS` nor the `CTRCFS` macros.)

#### 9.2.1 Priority Assignments

The handler prefix module defines global symbols for process and hardware priority, as follows:

Parameter	Definition
<code>xx\$PPR</code>	Process priority for controller process
<code>xx\$FPR</code>	Fork routine priority
<code>xx\$HPR</code>	Hardware interrupt priority
<code>xx\$IPR</code>	Initialization process priority

You should set the priority of the controller process relative to the hardware interrupt priority of the device it is controlling. Table 9-1 lists device-handler process priorities recommended by DIGITAL. (Process priorities, including those for device handlers, are listed in Appendix B of this manual.)

## GUIDE TO WRITING A DEVICE HANDLER

Table 9-1  
Recommended Device-Handler Process Priorities

Hardware Interrupt Request Level	Controller Process Priority (Range)	Recommended Initialization Process Priority (Range)
4	160-175	248-255 (250 recommended)
5	176-191	248-255 (250 recommended)
6	192-207	248-255 (250 recommended)
7	208-223	248-255 (250 recommended)

### NOTE

1. In the current version of MicroPower/Pascal, all initialization procedures in Pascal processes execute at software process priority 255. The initialization priorities listed in Table 9-1 are currently followed only by DIGITAL-supplied device handlers written in MACRO-11. This may change in a future release.
2. Hardware device interrupt requests at levels 4 to 7 are supported only by LSI-11/23 and SBC-11/21 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 set, interrupts are disabled; when cleared, interrupts are enabled.)
3. 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. Therefore, you should select process priority in accordance with hardware interrupt request level and device controller position along the bus for each interrupt request level listed.

## GUIDE TO WRITING A DEVICE HANDLER

Since the priority of the controller process managing the lowest-priority hardware device can be as low as 160, assign priorities between 1 and 159 to processes requesting I/O operations; that is, assign priorities lower than those given to device-handler controller processes.

### 9.2.2 DRVCFS Macro

DRVCFS, the first macro to be invoked, is the handler configuration macro and specifies the handler prefix and the number of controllers to be supported by the handler. DRVCFS defines global symbols used for configuration. The DRVCFS macro and its parameters are as follows:

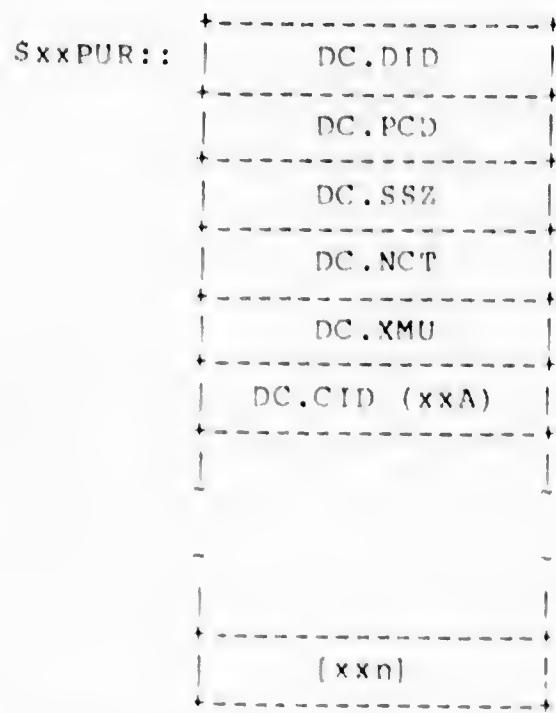
DRVCFS dname=xx,nctrl=n

Parameter	Definition
xx	Two-letter device identifier
n	Integer specifying the number of controllers that the device handler must support

When executed, DRVCFS invokes the device-handler impure-area definition macro (xxISZ\$), which is device-specific. DRVCFS 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 handler
xx\$MXU	Maximum number of units that can be supported by a single controller
SxxIMP	Address of the handler impure area
xx\$NUM	Number of controllers to be supported in this configuration
SxxPUR	Address of the handler configuration data; DRVCFS produces the data structure shown below for the device-handler initialization process; names prefixed with DC. are offsets from SxxPUR, not symbol names

## GUIDE TO WRITING A DEVICE HANDLER



Name	Definition
DC.DID	Two-letter device identifier (xx)
DC.PCD	Pointer to the device-handler process-creation data (SxxPCD)
DC.SSZ	Process stack size, in bytes (xxSSZ)
DC.NCT	Number of controllers supported by a single controller (xxSXMU)
DC.XMU	Maximum number of units supported by a controller (xxSMXU)
DC.CID	Pointer to the controller A initialization data; additional data words follow, one for each controller (B, C, ... n), as required

The invocation of the DRVCFS macro must always appear before the controller configuration macro, CTRCFS.

### 9.2.3 CTRCFS Macro

Separate invocations of the controller configuration macro (CTRCS) are required for each controller supported. Each CTRCFS macro specifies:

- The handler 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 handler being configured

## GUIDE TO WRITING A DEVICE HANDLER

The CTRCFS macro and its parameters are as follows:

```
CTRDFS cname,nunits,<csr1,vec1[,csr2,vec2...,csrN,vecN]>,units,  
<type1,xprm1[,type2,xprm2...,typen,xprmn]>
```

Parameter	Definition
cname	One-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 has no meaning for the KXT11-C Two-Port RAM (KX) handler, which allocates unit numbers, starting at 0, to KXT11-C dual-port RAM channels in the order that the CSR-vector parameter pairs are specified. Specify units=<> for the KX handler.

type1	Hardware type associated with the first CSR; hardware type codes for standard device handlers are defined in the file DRVDEF.MAC
xprm1	Extra parameter for the first CSR
typen	Hardware type associated with the nth CSR
xprmn	Extra parameter for the nth CSR

### NOTE

The KXT11-C TU58 (DD) handler uses the type-and-parameter pair to specify an asynchronous serial line type and a baud rate. However, for most handlers, the type-and-parameter pair is omitted.

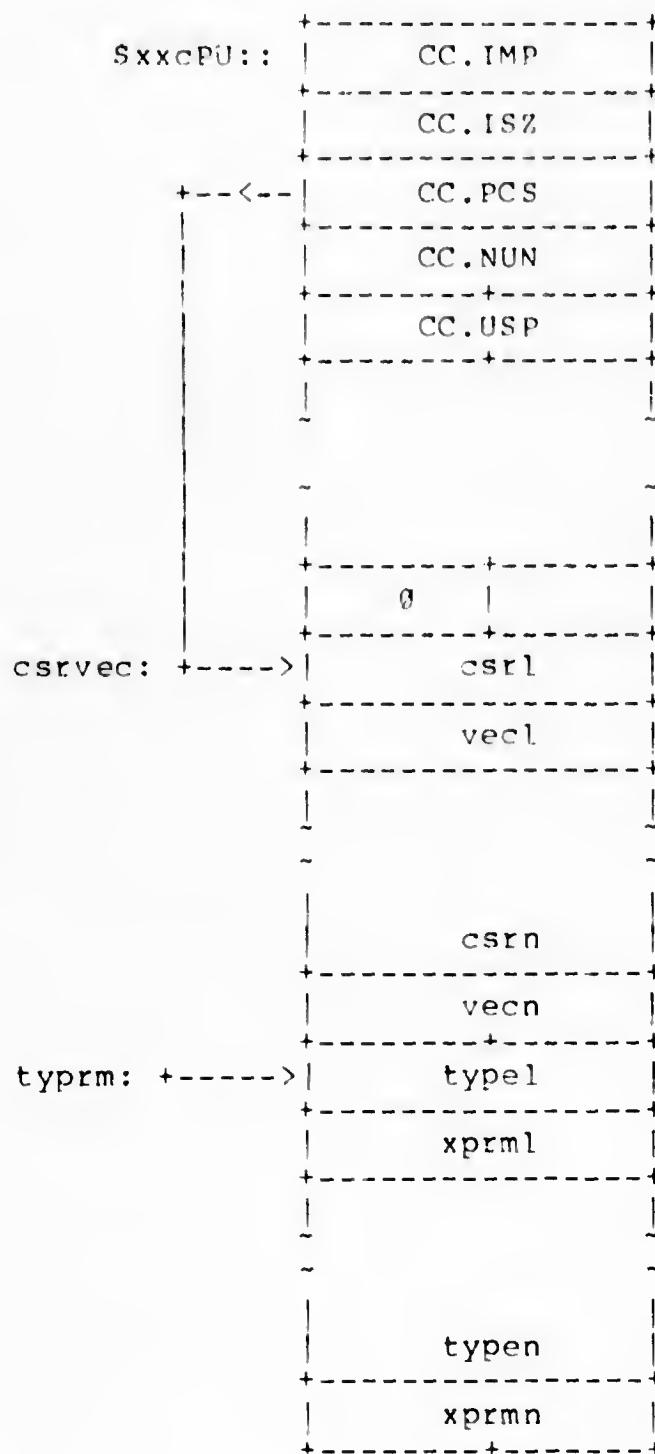
## GUIDE TO WRITING A DEVICE HANDLER

The CTRCFS macro, which is defined in the file DRVDEF.MAC, defines several symbols, as listed below. The xx characters in the symbols shown represent a 2-letter device identifier. The letter c in the symbols shown 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
\$xxcCPU	Address of the pure-code configuration data for controller c
\$xxcCS	CSR address for controller c
\$xxcVE	Interrupt vector address for controller c

CTRCFS produces the data structure shown below for the device-handler initialization process. Names prefixed with CC. are offsets from \$xxcCPU, not symbol names.

## GUIDE TO WRITING A DEVICE HANDLER



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 (impsz)
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

## GUIDE TO WRITING A DEVICE HANDLER

**csrvec**    A series of word pairs containing the CSR address and vector address for each unit

**typrm**    A series of word pairs containing the hardware type and extra parameter for each unit -- zero if not used by the handler being configured

CTRCS\$ generates the controller impure area, as follows:

```
$xxcIM::          .blk  xxSISZ ; Impure area  
                  .even  
$xxcG1::          .blk  xxSSSZ ; Low stack guard word  
                  .blk  xxSSSZ ; Stack space  
$xxcG2::          .blk  xxSSSZ ; High stack guard word
```

## GUIDE TO WRITING A DEVICE HANDLER

### 9.2.4 Sample Handler Prefix Module (DYPFX.MAC)

The following is an example of a handler prefix module.

```
.nlist
.enabl LC
.list
.title DYPFX - RX02 Device driver prefix module

; COPYRIGHT (c) 1982, 1983, 1984, BY DIGITAL EQUIPMENT
; CORPORATION. ALL RIGHTS RESERVED.

.mcall drvcfs
.mcall ctrcfs

DYSPPR == 184.          ; Process priority
DYSFPR == 184.*256.      ; Fork routine priority
DYSHPR == 5              ; RX02 hardware priority
DYSIPR == 250.           ; Process initialization priority

drvcfs dname=DY,nctrl=2
ctrccfs cname=A,nunits=2.,csrvec=<177170,264>,units=<0:1>
ctrccfs cname=B,nunits=2.,csrvec=<177200,270>,units=<0:1>

.end
```

### 9.3 DEVICE-HANDLER IMPURE-AREA DEFINITION MACRO (xxISZ\$)

The device-handler impure-area definition macro defines configuration parameters required by the device-handler prefix module. When adding a custom device handler to the system, add a device-handler impure-area definition macro to the DRVDEF.MAC library. The macro should include the following parameters:

Parameter	Definition
xxSISZ	Number of bytes needed for the fixed part of the impure area by a controller process
xxSSSZ	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 MISDFS macro). For example:

```
xxSSSZ = $MINST+n
```

In the example above, n is the number of bytes on the process stack needed by the process itself.

## GUIDE TO WRITING A DEVICE HANDLER

**xx\$USZ** Number of additional bytes needed for each unit supported by the handler

**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 handler proper. When these values change through program modification, edit the **xx\$ISZ\$** macro to reflect the changes. For example, the impure-area definition macro (**DDISZ\$**) for the TU58 (DD) device handler defines configuration parameters as follows:

```
.MACRO DDISZ$  
DD$USZ = 0  
DD$SSZ = 200.  
DD$ISZ = 150  
DD$MXU = 2.  
.ENDM DDISZ$
```

### 9.4 DEVICE HANDLER PROPER

The device handler proper is the device-handler source module. (The device-handler prefix module and the impure-area definition macro previously described provide configuration information required by the handler proper.) The following paragraphs describe the composition of the device-handler source module.

By convention, a device-handler source has a file name of the form **xxDRV.MAC**. In general, device handlers supplied by DIGITAL in the MicroPower/Pascal distribution kit conform to MicroPower/Pascal and MACRO-11 coding conventions. You can use the RX02 device-handler MACRO-11 source file, **DYDRV.MAC**, as an example when you write device handlers in MACRO-11; a listing of **DYDRV.MAC** is contained in Appendix A.1 of this manual.

When writing device handlers in Pascal, refer to the **YADRV.PAS** source code for the **DRV11** device handler; a listing of **YADRV.PAS** is contained in Appendix A.2 of this manual.

A device handler written in MACRO-11 is divided into several discrete sections:

1. Copyright page
2. Module header
3. Functional description
4. Local macro definitions
5. Externally defined symbol defaults
6. Private and/or local data and symbol definitions, including:
  - a. Impure area common to all copies of the handler
  - b. Pure data tables

## GUIDE TO WRITING A DEVICE HANDLER

7. An initialization process that creates the required data structures, initializes impure areas, and starts the controller processes
8. An impure area that contains state information for the controller processes
9. A constant data area that contains constants and tables
10. One or more controller processes that wait for I/O requests, activate the device, and wait for a response or a timeout
11. One or more interrupt service routines that respond to interrupts from the device
12. One or more fork routines that signal the controller processes when the device requires further attention
13. A reply routine that returns completion status -- successful completion or error code -- to the process that requested the I/O operation
14. A termination procedure that deallocates owned resources previously obtained by the controller process before deleting the controller process
15. Error-processing routines that:
  - a. Retry on recoverable errors
  - b. Report nonrecoverable errors to the requesting process
  - c. Return to the pool I/O request packets that cannot be returned to the requestor

Handler components are discussed in detail in the following sections.

### 9.4.1 Copyright Page

The copyright page, although optional, 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 a .title and a .ident directive. These will 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 handler  
.ident /V01.00/
```

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 (;).

## GUIDE TO WRITING A DEVICE HANDLER

### 9.4.2 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.

### 9.4.3 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 handler as I/O requests are processed and any other design interface information needed for using the device handler in an application.

### 9.4.4 Declarations

Declarations include system macro requirements, local macro definitions, external symbol definitions -- global symbols defined in the handler prefix module -- and local symbol definitions, as required. The following sections describe the declarations.

**9.4.4.1 Local Macro Definition** - All macro definitions for local macros referenced by the device handler are contained in this section. (Local macros are referenced only by the device handler proper.)

**9.4.4.2 Externally Defined Symbols** - The section of externally defined symbols may include symbols containing default values. You need the macros listed when assembling the device-handler module. Default values for the macros can be changed, as required, by device-handler code. For example, a macro may state the number of retries for read request processing. The particular device handler may also process read-with-no-retries requests, requiring the default value modification.

**9.4.4.3 Process Definition** - The device-handler 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 DFSPCS kernel primitive (described in Chapter 3). All parameters except ini are required. The following example is the process definition for the DY device handler:

```
DFSPCS pid=$DYADR,pri=DY$IPR,typ=PT.DRV,cxo=0,grp=1,ter=DYDO,  
cxl=0,stl=$DYAG2,stl=$DYAG1,sth=$DYAG2,start=DYINIT,ini=0
```

Parameter	Definition
pid	\$DYADR, the name of the static process (DY handler, controller A)
pri	DY\$IPR, the initialization-process priority specified in the handler prefix module

## GUIDE TO WRITING A DEVICE HANDLER

typ	PT.DRV, indicating device handler (driver) mapping
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	DYDO, the DY device-handler 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 handler must request a dynamic process if the device handler supports multiple controllers and if the common device-handler initialization routine \$DDINI is used in the handler's initialization process. You request a dynamic process by issuing the CRPC\$P kernel primitive (described in Chapter 3). The following example is the dynamic process request for the DY device handler:

```
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 handler 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
stl	0 (no source of low boundary address of process stack); \$DDINI will supply a value of \$xxcG1 for this parameter

## GUIDE TO WRITING A DEVICE HANDLER

sth	0 (no source of high boundary address of process stack); \$DDINI will supply a value of \$xxcG2 for this parameter
start	DYSTART, the controller process entry address
ini	0 (null parameter)

**9.4.4.4 Impure-Area Definition** - The impure area is the writeable data area for the device handler. Each copy of the controller process has its own impure area. Additional space may be allocated in a section according to the number of handlers supported by the controller process. You can see typical contents of the impure area by examining the DY device-handler source. Note that the request semaphore SDB is the first item in the impure area, that the stack follows the impure area, and that the ISR impure area is also contained within the impure-area definition.

**9.4.4.5 Pure-Area Definition** - The pure-area definition is a read-only area that is shared by all handler processes. This area typically contains text, tables, and data that never requires modification. In a mapped system, the protection for this area is read-only. For example, in the DY device handler, this area includes the action routine dispatch table, device characteristics table, and table of error-return codes common to all controller processes.

### 9.4.5 Initialization Process

The initialization process typically creates one or more queue semaphores by which the requesting processes communicate with the device handler, start the handler processes for each controller, and initialize the impure area. The handler-initialization process is a statically defined process that executes at very high priority at start-up time. Priority values 248 to 255 are reserved for start-up initialization; 250(decimal) is recommended. A range of priorities permits specific initialization processes to execute before others, as required.

A common device-handler initialization routine, \$DDINI, simplifies initialization-process coding for handlers written in MACRO-11. 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 handler performs its functions by calling the \$DDINI routine. The required input data is contained in \$DYPUR; a pointer to \$DYPUR is passed in R5. (\$DYPUR is as described in Section 9.2.2.) For example:

```
DYINIT::           ; Initialization entry point
    MOV      #$DYPUR,R5
    CALL    $DDINI          ; Data is returned on the stack
```

## GUIDE TO WRITING A DEVICE HANDLER

SDDINI exits with the stack containing the controller ID and pointers to the impure area and initialization data, as listed below. The controller processes, normally entered immediately following SDDINI execution, must, immediately on entry, remove this controller-specific information from the stack for use by the handler code.

- SP+04 Controller ID (0=controller A, 1=controller B, and so forth)
- SP+02 Pointer to initialization data (\$xxcPU, as described in Section 9.2.3)
- SP+00 Pointer to controller's impure area

### 9.4.6 Controller Process

The controller process performs the following functions:

1. Reads handler requests
2. Validates the requests
3. Initiates the I/O functions
4. Waits for I/O completion or device timeout and returns status to the requestor

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 handler process. This is done by lowering the process priority to that of the controller process. In the case of multiple handler processes, the initialization process becomes the first handler process and creates copies for each additional process. (SDDINI performs this function.)

The DY device-handler controller process includes several routines and subroutines that prepare the handler for I/O, receive and validate I/O requests (queue server), and execute the I/O requests (interrupt procedure, action routines, and so forth). An overview of the major routines and their functions is provided below.

The DY handler controller process entry point is DYSTRT. Upon entry, the controller process removes and saves the pointers passed by SDDINI; however, it discards the controller ID. The controller process then creates an unnamed binary semaphore that will connect the handler's ISR to the I/O request service process and will connect the interrupt vector to the ISR.

The Request Process Queue Server (DYREQ) is the device handler's main request-processing code. DYREQ performs the following functions:

1. Sets up retry count for request
2. Validates unit number specified in request; returns a . error message if invalid
3. Checks function code specified in request; returns an error message if invalid

## GUIDE TO WRITING A DEVICE HANDLER

4. Copies device characteristics to the queue element for message returned to the process requesting the I/O operation
5. Performs device-specific operations and calls the interrupt procedure (subroutine)
6. Returns an appropriate completion status message to the process requesting the I/O operation once the I/O operation is completed

Action routines (subroutines) include: WF, WW, RE, RR, and ZF. These five routines are called, as appropriate, when processing read, write, and zero-fill operations with the RX02 hardware.

The Interrupt Procedure (DYINT) performs several functions, as follows:

1. Waits for an interrupt signaling a completed RX02 operation
2. Saves the function just completed for possible retries
3. Calls the next function that must be performed with the RX02, if no errors were detected
4. Executes retry -- repeats the saved function -- if appropriate, if errors were detected; otherwise, returns an error status to the caller

### 9.4.7 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 8. 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:

1. Very limited context
2. Very brief execution periods
3. Restricted processing capabilities

In a mapped system, an ISR is mapped to the kernel, the system-common area, and the I/O page. Map the ISR to the kernel via CPU kernel PARS 2 (code) and 3 (data) by specifying PT.DRV for the typ parameter in the DFSPCS primitive (see Section 9.4.4.5). When relocating the merged object modules -- part of building the application -- map the handler code, ISR, and impure area by specifying the /O:40000/X option in the RELOC utility command string. (Refer to 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.)

When an ISR is called, the kernel sets the CPU priority to the value specified in the CINTS primitive issued by the device handler. 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.

## GUIDE TO WRITING A DEVICE HANDLER

Before issuing any kernel primitive, the ISR must go to fork-level processing. This is done by issuing a FORK\$ call. Once 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 handler (\$DYDUM). Upon entry, the ISR immediately issues a FORK\$ request. Once at fork level, it signals the Interrupt Procedure via a SGNLSS kernel primitive, indicating completion of the requested operation. Since the DY device is a DMA device, all data transfers have been completed, and the interrupt simply informs the device handler of that fact.

Other device handlers supplied in your MicroPower/Pascal distribution kit contain more extensive ISRs. For example, the ISR in the TU58 device handler (DDDRV.MAC) is a priority-7 ISR that performs single-character I/O. In addition, the SBC-11/21 PIO port device handler (YFDRV.MAC) includes a normal ISR that contains internal queueing. You should carefully examine those ISRs for programming techniques to use in your application.

### 9.4.8 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. Hence, kernel primitives generally 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 process level.

Each FORK\$ call includes a parameter, defined in the handler prefix module, that specifies fork queue priority for the fork routine that follows, relative to other fork routines in the system. Fork-routine priority should be established relative to hardware interrupt priority. (Refer to the description of the FORK\$ call in Chapter 3 for a complete description of the fork process and fork routine priority.)

#### NOTE

Fork-routine priority for DIGITAL-supplied handlers is generally equal to the controller-process priority multiplied by 256(decimal). For example, the DY fork-routine priority (DY\$FPR) is defined as 184.\*256.

### 9.4.9 Reply Subroutine

The reply subroutine returns completion status to the process that requested the I/O operation. Status will indicate either successful completion or an appropriate error code.

## GUIDE TO WRITING A DEVICE HANDLER

Upon entry, the reply subroutine should receive the status code from the caller, a pointer to the message packet, and a pointer to the impure area. The reply subroutine inserts the status, error code, and byte count involved in the I/O operation in the reply queue element. For example, the DY handler return-status message subroutine (REPLY) receives a message packet from a calling routine, sends the message via the SGLQSS kernel primitive, and returns to the caller.

### 9.4.10 Termination Procedure

Each device handler includes a termination procedure that is entered whenever the device handler is stopped (aborted). The termination procedure deallocates, in an orderly manner, all resources that the handler had previously acquired. The termination procedure performs the following functions:

1. Disables interrupts on the appropriate controller hardware
2. Cancels any active timeout requests
3. 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
4. Destroys all structures created by the handler
5. Deletes all processes created by the handler
6. Deletes the handler controller process

For an example of a termination procedure, refer to the DY handler Request Process Termination Procedure (DYSTP).

### 9.4.11 Error-Processing Routines

The handler processes several categories of errors, as follows:

1. Invalid request packets
2. Exceptions (traps -- memory timeouts, illegal instructions, and so forth)
3. Drive or controller hardware errors
4. Resource famine (insufficient kernel pool to allocate structures, and so forth)

**9.4.11.1 Invalid Requests** - Handlers 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 handler returns the invalid packet to the sender. If that is not possible, the packet must be sent to the error-recording process with status information indicating an invalid request.

## GUIDE TO WRITING A DEVICE HANDLER

**9.4.11.2 Exceptions** - Exceptions are memory timeouts, illegal instructions, traps, and so forth, as follows:

1. 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
2. One or more programming errors in the device handler
3. A hardware failure -- CPU error, memory error, or controller error
4. Nonexistent hardware

Device handlers 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 that their occurrence does not cause unexpected runtime 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 runtime errors.

Applications should include exception handlers for all possible types of exceptions that may be caused by processing bad data in a request packet. For example, if an incorrect address supplied in the request packet causes the handler to generate a memory fault (trap to 4), the handler should provide a memory fault exception handler to deal with the problem. (Exception handling and dispatching are discussed in Chapter 7.)

**9.4.11.3 Drive or Controller Errors** - Drive or controller errors can be either recoverable or nonrecoverable. In the case of recoverable errors, the device handler 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 handler returns an appropriate error status to the requesting process and error-recording process.

**9.4.11.4 Resource Famine** - If a standard device handler or the common initialization routine \$DDINI cannot obtain enough memory to create a process or structure, it will report an exception by calling the common exception-reporting routine \$DDEXC (see Section 9.4.11.5), which issues an SREXC kernel primitive request. Resource famine errors should occur only during the debugging stages of application program development.

**9.4.11.5 Exception-Reporting Routine (\$DDEXC)** - A common device-handler routine, \$DDEXC, simplifies exception reporting for handlers written in MACRO-11. The \$DDEXC routine is called to report an exception, usually immediately after a CINT\$, CRPCS, CRSTS\$, or other kernel primitive request has returned an error. After the exception is reported, the calling process is stopped. (There is no return from the call.)

## GUIDE TO WRITING A DEVICE HANDLER

The \$DDEXC routine has 10 unique entry points, one for each handler exception that \$DDEXC can declare, and 3 group entry points, for CINT\$, CRPCS, and CRSTS failures, respectively. No input parameters are associated with the unique entry points. For the group entry points, R0 must contain the error code returned by the primitive (\$CINT, \$CRPC, or \$CRST) that failed. A group entry point dispatches to the appropriate unique entry point based on the error code in R0. At each unique entry point, an exception is declared with a specific exception type and subcode (see below).

The entry points for the \$DDEXC routine are as follows:

### Unique Entry Points

Entry	Type	Subcode	Description
\$DECIB	EXSSVC	ESSAOV	CINT\$ failures: Already owned vector
\$DECIA	EXSSVC	ESSIAD	Invalid address
\$DECIV	EXSSVC	ESSIVC	Illegal vector
			CRPCS failures:
\$DECIM	EX\$RSC	ESSNMK	Insufficient space for kernel structure
\$DECIN	EX\$SVC	ESSSIU	Structure is in use
			CRSTS failures:
\$DESTM	EX\$RSC	ESSNMK	Insufficient space for kernel structure
\$DESTP	EXSSVC	ESSIPM	Illegal parameter
\$DESTI	EXSSVC	ESSSIU	Structure is in use
			Structure access failure:
\$DESID	EXSSVC	ESSIST	Illegal structure identifier
			Miscellaneous failures:
\$DEICD	EXSSIO	ESSICD	Invalid configuration data

### Group Entry Points

Entry	Primitive	Description
\$DEGCI	CINT\$	Connect to Interrupt failures
\$DEGCP	CRPCS	Create Process failures
\$DEGCS	CRSTS	Create Structure failures

Both of the MACRO-11 device handlers listed in Appendix A use the \$DDEXC exception-reporting routine.

**APPENDIX A**  
**SAMPLE DEVICE HANDLERS**

Source codes for three device handlers are included in this appendix as examples. The first two handlers (DY and XL) are MACRO-11 examples. The third handler (YA) is a Pascal example.

In this appendix, long macro invocations are continued on a second line. When writing source code in MACRO-11, however, you must have the entire invocation on one line.

## SAMPLE DEVICE HANDLERS

### A.1 RX02 (DY) HANDLER--SAMPLE MACRO-11 DEVICE-HANDLER SOURCE

```
.nlist           ;Edit Level 18
.enabl  LC
.list
.title  DYDRV - RX02 Driver
.ident  /V01.00/
;
;          COPYRIGHT (c) 1982 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 WHICH IS NOT SUPPLIED BY DIGITAL.
;+
;
;          Module name: DYDRV.MAC
;
;          System: MicroPower/Pascal
;
;          Author: BJH           Creation date: 12 Sep 80
;
;          Modified:
;
;
;          Functional Description:
;
;          This module provides device driver services for the RX02 flexible
;          diskette. It controls any number of RX02 drives. It supports only
;          double density operations and will not read or write a single density
;          floppy. It will, however, reformat a single density floppy to double
;          density.
;
;
;- .sbttl Declarations
;+
;
;          System macro requirements
;
;
;.enabl  GBL
.mcall  macdf$, iodf$, quedf$, drvdf$, dcdf$, dyisz$  

macdf$  

iodf$  

quedf$  

drvdf$  

dcdf$  

ccdf$  

dyisz$
```

## SAMPLE DEVICE HANDLERS

```
.mcall cint$, crpc$p, crst$, dapk$, dint$, dlpc$, dlst$  
.mcall fork$, sgnl$, sglq$, waiq$, waqc$, xtad$  
  
;  
;  
; Local macro definitions:  
;  
;-  
  
;  
;  
; External symbols which are necessary to assemble this module but  
; which may have default definitions:  
;  
;-  
  
dfalts DYSRTY,10. ; Retry count  
  
;  
;  
; External symbols defined in the driver prefix module.  
;  
;-  
  
.globl DY$FPR ; DY fork process priority  
.globl DY$HPR ; RX#2 hardware priority  
.globl DY$PPR ; DY process 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: .blk b SD.SIZ ; Request semaphore SDB  
UNTMAP: .blk b ; Unit allocation bit map. Only 2 bits  
ADTYPE: .blk w ; required since only 2 units per controller  
; Physical or logical (also  
; used to indicate first sector  
; of a track for DADINC):  
; 0 physical,1 logical,-1 first  
RETRYs: .blk w ; Number of retries left  
RTYCNT: .blk w ; Total number of retries (0 if RTYINH)  
LASTSC: .blk w ; Last (interleaved) sector  
; on a track (used by DADINC)  
UNIT: .blk w ; Density/unit number for command  
PACKET: .blk w ; Pointer to request packet  
ERROR: .blk w ; Error code  
STATUS: .blk w ; Status code  
LASTFU: .blk w ; Address of last sub function  
SECTOR: .blk w ; Sector to be accessed  
BYTES: .blk w ; Bytes to be xfered this sector  
TRACK: .blk w ; Track to be accessed  
BOUND: .blk w ; Last odd sector in track  
EXTADR: .blk w ; Extended address bits for buffer
```

## SAMPLE DEVICE HANDLERS

```

BUFFER: .blkw          ; Buffer address of user buffer
NEXTFU: .blkw          ; Next subfunction to be executed
DONE:   .blkw          ; Need to zero fill flag
                  ; zf if zero filling required
                  ; 0 if no zero filling required
LSTBYT: .blkw          ; Byte count of last transfer
RX2CSR: .blkw          ; Address of RX02 CSR
RX2DB:  .blkw          ; Address of RX02 Data Buffer
VECTOR: .blkw          ; Interrupt vector
$DYINT: .blkw          ; Address of control semaphore SDB
ISRIMP:              ; ISR impure area
CNTSEM: .blkb SD.SIZ  ; Control semaphore sdb
                  .even
..ISZ   =   . - .temp. ; Size of impure area.
G1:    .blkw          ; Low stack guard word
STKBOT: .blkb DYSSS2  ; .even
                  ; High stack guard word

; Validate parameters specified in the dyisz$ macro

.if LT <DY$ISZ - ..ISZ>
    .error DY$ISZ ; Impure area too small as specified in DY$ISZ$
    .error ..ISZ  ; Edit DY$ISZ$ 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+70>>
    .error DY$SSZ ; Stack size specified in DY$SSZ is too small
.endc

.if GT DYSUSZ
    .error DYSUSZ ; Value specified in DY$SSZ is unnecessary
.endc

; RX02 Device Register Definitions:

; Control and Status Register Bit Definitions

CSGO   =   1           ; Initiate function
CSUNIT =   20          ; Unit bit
CSUN0   =   0           ; Unit zero
CSUN1   =   20          ; Unit one
CSDONE =   40          ; Done bit
CSINT   =   100         ; Interrupt 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
CSINIT  =   40000        ; RX11 initialize
CSERR   =   100000       ; Error
DENERR  =   20           ; RXDB density 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
CSR DST = 5*2          ; 5 - Read status
CSWRD  = 6*2           ; 6 - Write sector with deleted data
CSMAIN = 7*2           ; 7 - Maintenance

```

## SAMPLE DEVICE HANDLERS

```
assum$ CSRD2 NE 0 ; 2 bit must be on in read
assum$ CSWRT2 EQ 0 ; 2 bit must be off in write
assum$ CSWRTD2 EQ 0 ; 2 bit must be off in write

; Error and Status Register Bit Definitions

ESCRC = 1 ; CRC error
ESSIDI = 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)
SECSIZ = 256. ; Number of bytes/sector double density

.sbttl Process definition

dfspc$ pid=$DYADDR,pri=DY$IPR,typ=PT.PRV,cxo=0,grp=1,ter=DYDO,cxl=0,
      sti=$DYAG2,stl=$DYAG1,sth=$DYAG2,start=DYINIT,ini=0

pdat$

$DYPCD:::
    crpc$p pdb=PDB,pri=DYSPPR,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:

FTABLE: .word RR ; Read physical
        .word RR ; Read logical
        .word 0000 ; Reserved for future use
        .word WF ; Write physical
        .word WF ; Write logical
        .word 0000 ; Reserved for future use

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
```

## SAMPLE DEVICE HANDLERS

```
; Device Characteristics Table:  
  
DTABLE: .byte DCSDSK ; Class is disk  
.byte DKSDY2 ; 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:  
    ISS$CTL ; Return controller error if no error bits set  
    ISS$PAR ; Parity error for CRC error  
    0 ; Nothing for Side 1 ready  
    ISS$CTL ; Controller error for initialize done  
    ISS$PWR ; Power fail for RX power failure  
    ISS$IVM ; Invalid mode for density error  
    0 ; Nothing for drive density  
    0 ; Nothing for deleted data mark  
    ISS$UNS ; Unsafe for drive not ready  
    0 ; Nothing for unit selected  
    0 ; Nothing for reserved bit  
    ISS$OFL ; Word count overflow during fill or empty  
    ISS$NXM ; Non-existent memory  
  
    pure$  
.sbttl DYDO - Process termination routine  
;+  
;  
; DYDO - Process termination routine  
DYDO:  
    dlpc$ ; Terminate the init process  
  
.sbttl DYINIT - Power Up Initialization Process  
;+  
;  
; DYINIT - This process creates an instance of the device  
; driver process for each controller 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 terminates itself after initialization  
; is complete.  
;  
;-  
  
DYINIT::  
    MOV     #SDYPUR,R5 ; -> Configuration data for RX02  
    CALL    SDDINI ; Common device driver initialization  
                ; routine  
.sbttl I/O Request Service Process  
.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  
;  
;-
```

## SAMPLE DEVICE HANDLERS

```

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,$DYINT(R5)      ; Save pointer to SDB

    crst$S sdb=R2, styp= ST.BSM, 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),R3            ; R3 -> RX02 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= $DYDUM,imp=R2,pic= 0

10$:   BCC      30$            ; Br if no error
10$:   dlst$S R2             ; Init failure, delete semaphore
20$:   dlpc$                ; Then terminate this process

30$:   push$   #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    SUNTMAP          ; Set bitmap for supported drives
      TST     (SP)+            ; Any errors?
      BNE     DYREQ            ; Br if not
      CALL    SDEICD           ; Report error (Does not return)

.sbtll DYREQ - Request Process Queue Server
;+
;
; This is the I/O request service process. It receives all queued
; I/O requests, checks them for legal function codes, sets up the
; function to be executed (calling DAD and BUFFAD ) and initiates
; the transfer.
;
;      R5 -> Impure area (at request semaphore SDB)
;      R4,R1  Modified
;
;-
;

.enabl LSB

DYREQ:
    waiq$S R5,R4            ; Wait for a request packet
    BCC      5$              ; Br if no error
    CALL    $DESID           ; Report exception (Does not return)

5$:    assum$ REQSEM EQ 0
      MOV      R4,PACKET(R5)    ; Save -> packet in impure area
      xtad$  DP.BUF(R4),DP.PAR(R4),pos=12.,ext=EXTADR(R5),addr=BUFFER(R5)

```

## SAMPLE DEVICE HANDLERS

```

; Set up retry count, if not inhibited by function modifier

    MOVB    #DY$RTY, RETRYS(R5)      ; Set up retry count.
    BIT     #FM$INH, DP.FUN(R4)      ; See if retry is inhibited.
    BEQ    10$                      ; If EQ, no, retry if errors
    CLRB    RETRYS(R5)              ; Else inhibit retry on error
10$:   MOVB    RETRYS(R5), RTYCNT(R5) ; Set retry count= total retries

; Validate unit number (ALL RXO2'S HAVE MAX. OF TWO DRIVES)

    MOV     #CSUN1!CSDN, UNIT(R5)   ; Assume unit one is wanted.
    push$   R5                     ; Compute -> unit bit map
    ADD     #UNTMAP, (SP)          ;
    CLR     -(SP)                 ;
    MOVB    DP.UNI(R4), (SP)       ; Clear to convert byte to word
    BNE    20$                    ; Specified unit number
    MOV     #CSUN0!CSDN, UNIT(R5)   ; Br if not unit zero
    20$:  push$    #DY$MXU         ; It is unit zero
          CALL    $UNTVA           ; Maximum number of units
          TST     (SP)+             ; Validate unit number
          BNE    30$                 ; Unit number valid?
          MOV     #IS$NXU, STATUS(R5) ; Br if so
          BR     50$                 ; Else nonexistent unit error
                                ; Return message/process next request

30$:  CLR     ERROR(R5)          ; Clear error flags
    CLR     STATUS(R5)           ; and status flags

; Check function codes 0,1=READ 3,4=WRITE 2,5=ILLEGAL 6=SET, 7=GET, 10+=ILLEGAL

    MOV     DP.FUN(R4), R1          ; R1 = function code and modifiers
    BIC     #^C77,R1               ; Clear all but function bits
    CMP     R1, #IF$GET            ; Check for legal range of code
    BHI    40$                    ; Br if illegal function
    MOVB    PLTABLE(R1), R0        ; Read or write function?
    BPL    60$                    ; Br if so
    ASLB    R0                    ; Get or set characteristics?
    BEQ    40$                    ; Br if not, a reserved code which
                                ; we treat as illegal

    assum$ <IF$GET1> EQ 1
    assum$ <IF$SET1> EQ 0
    ASR     R1                    ; Get or set characteristics
    BCC    40$                    ; Br if set, which we don't support

; Copy device characteristics to the queue element

    MOV     #DTABLE, R0            ; -> Device characteristics table
    MOV     R4, R1                ; -> Queue element
    ADD     #DP.DAD, R1            ; -> Device characteristics block
    MOV     #DTBSIZ, R2            ; = Number of bytes in table
    35$:  MOVB    (R0)+, (R1)+      ; Copy data
    SOB     R2, 35$               ; Loop until done
    BR     50$                    ; Return message/process next request

40$:  MOV     #IS$FUN, STATUS(R5) ; Return illegal function code
50$:  CALL    REPLY              ; No, just signal user
    BR     DYREQ                 ; Process next request

60$:  CMPB    R1, #IF$WTP          ; Physical write?
    BNE    70$                    ; Br if not
    BIT     #FM$WFM, DP.FUN(R4)    ; Format disk?
    BEQ    70$                    ; Br if not

```

## SAMPLE DEVICE HANDLERS

```

; Format the diskette in double density mode

    CMP      #"FO,DP.DAD(R4)           ; A safety check to verify that the
    BNE      40$                   ; user really wants to format the disk
                                    ; Br if he really didn't want to do it
    CLR      STATUS(R5)            ; Return illegal function code
    MOV      #CSFMT!CSDN!CSGO,R2   ; No errors possible
    BIS      UNIT(R5),R2          ; Set up format command for exec2
    MOV      #111,R0               ; Set unit number
    CALL    EXEC2                 ; Fail safe character
    BR      50$                   ; Do it
                                    ; Return message/process next request

; Check to see if we will need to zero fill the last half of the last block.
; If so, we replace done with zero fill.

70$:   MOV      DP.LEN(R4),R2        ; R2 = Number of bytes
       MOV      R2,BYTES(R5)        ; Save number of bytes
       CLR      DONE(R5)           ; Assume even number of sectors
       SWAB    R2                  ; Divide # of bytes by 256 (sector size)
assum$ SECSIZ EQ 256.
       TSTB    BYTES(R5)           ; Do we have a partial sector?
       BEQ    80$                  ; No.
       INC      R2                  ; Yes, include it in the
                                    ; number of sectors.

80$:   ASR      R2                  ; See if sector count is odd
       BCC    90$,                ; If CC, sector count is even
       MOV      #ZF,DONE(R5)        ; Else zero the last (odd) sector

; Set up transfer...

90$:   MOVB    PLTABLE(R1),ADTYPE(R5) ; See if this is physical or logical.
       ASL      R1                  ; Calc word offset for function table.
       MOV      FTABLE(R1),NEXTFUNC(R5); This is first function to execute.
       CALL    DAD                 ; Calculate device address
       BCC    100$                ; If CC, device address OK
       MOV      #ISSIBN,STATUS(R5)  ; Else return invalid block number
       BR      50$                  ; Return message/process next request

100$:  CALL    @NEXTFU(R5)          ; Do it.

       CALL    DYINT                ; and wait till error or done
assum$ ISSNOR EQ 0
       TST      STATUS(R5)          ; Was there an error we wish to
                                    ; Investigate ?
       BEQ    50$                  ; Reply and return if not

; This routine reads the extended error code from the RX02
; and places it in the error word to be shipped back to the user

       MOV      #CSMAIN!CSGO,R2     ; Read error reg, interrupts disabled
       MOV      R4,R0                ; -> Queue element
       ADD      #DP.FDD,R0          ; R0 -> Function dependent data
       CALL    EXEC2                 ; Go do it.
       MOV      ADTYPE(R5),ERROR(R5); Put RX02's response in 'ERROR'.
       BR      50$                  ; Return message/process next request

.dsabl LSB
.sbttl DYSTP - Request Process Termination Procedure

; 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:
       BIC      #CSINT,@RX2CSR(R5)  ; Disable further interrupts
       MOV      #IS$ABT,STATUS(R5)  ; Send abort status code to
       CALL    REPLY                 ; all waiting processes?

```

## SAMPLE DEVICE HANDLERS

```

CONT:    waqc$ s R5,R4
        BNE      DYSTP
TERM:    dlist$ s R5                      ; Delete all structures
        dlist$ s $DYINT(R5)          ; created.
        dint$ s VECTOR(R5)         ; Disconnect from interrupt vector
        dlpc$ 
.sbtll  DYINT - Interrupt Procedure
;+
;
; DYINT - This procedure is called to wait for an interrupt caused by
; the initiation of an operation. The interrupt service procedure runs
; in process context when signalled by the ISR. This means one sector
; has been transferred or an error has occurred. It checks for errors,
; retries soft errors, flags hard errors for the request process, and if
; no error has occurred initiates the next operation.
;
;      R5 -> Impure area
;
;      CALL      DYINT
;
;      R0 modified.
;      STATUS(R5) = Results of current function
;
;-
;
.dsabl  LSB
DYINT:
        wait$ s $DYINT(R5)          ; Wait for an interrupt signal
        BCC      5$                  ; Br if no error
        CALL     $DESID              ; Report error (Does not return)
;
5$:     TST      @R3                ; Any errors ?
        BLT      20$                ; If LT, yes
;
; Save the function just executed in case we need to retry it.
;
10$:   MOV      NEXTFU(R5),LASTFU(R5)    ; Set up retry in case of error
        BEQ      60$                ; Return if no next function
        CALL    @NEXTFU(R5)          ; Call next function
        BR      DYINT
;
; Error handler
;
20$:   BIT      #CSDONE,@R3           ; See if status is readable
        BEQ      20$                ; Wait till it is
        BIT      #ESCRC,@RX2DB(R5)  ; Is it a CRC error ?
        BEQ      30$                ; If EQ, no, it's a hard error
        DECB    RTYCNT(R5)          ; Decrement retry count
        BLE      30$                ; If LE, exhausted or inhibited
        CALL    @LASTFU(R5)          ; Otherwise, repeat last function
        BR      DYINT
;
; Hard error (or retries inhibited), retries exhausted or inhibited.
;
30$:   push$   @RX2DB(R5)            ; Copy error/status register
        MOV      ESDRDY,R0          ; Complement ready bit so that bit set
        XOR      R0,(SP)             ; indicates drive not ready
        MOV      #ERLIST,R0          ; -> list of status codes for each error
        BIC      #ESSID1!ESDDEN!ESDDAT!ESUNIT!ESHEAD,(SP);Clear bits which don't
                    ; indicate errors
;
40$:   BEQ      50$                ; 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      40$                ; Br if not right error
;
50$:   TST      (SP)+              ; Remove remnant of status register
        MOV      (R0),STATUS(R5)      ; Copy status indicator
60$:   RETURN

```

## SAMPLE DEVICE HANDLERS

```
.sbttl $DYDUM - Interrupt Service Routine (ISR)
;+
;
; $DYDUM - Interrupt Service Routine
;
; This code is invoked whenever an interrupt is received from the RX02.
; It simply signals the semaphore that DYINT is waiting on and returns.
; The ISR must be PIC code, so AMA is disabled here.
;
;-
.dsabl AMA
$DYDUM: FORKS #DY$FPR ; Request fork process at priority DY$FPR
        BCS 10$ ; If CS, previous fork not processed
        SGNL$S R3 ; Signal arrival of interrupt
10$:   RETURN ; Dismiss the real interrupt
.enabl AMA
.sbttl Interrupt Process Action Routines:
;+
;
; The interrupt process DYINT calls one of the five action routines
; WW, WF, RR, RE or ZF. The action routine called then calls
; the subroutines actually do the work.
;
;-
.sbttl WF - Fill Silo Write Action Routine
;+
;
; Write Fill- This routine fills the silo before a write. Sets the next
; subfunction to be write write (WW) unless the word count is exhausted
; when it sets the next function to be done.
; It also increments device address and buffer pointer when the word
; count is not exhausted.
;
;      R5 -> Impure area
;      R3 -> RX02 CSR
;      R2 = RX02 function code
;      R0 = Either word count or sector
;      R1 = Either buffer address or cylinder
;
;      CALL WF
;
;-
WF:   CALL WORDCT ; Sets up number of words to xfer
      MOV #CSFBUF!CSGO!CSINT,R2 ; Set fill code for execute routine
      MOV BUFFER(R5),R1 ; R1 = buffer address
      CALL EXECUT ; Execute the function
      INCB BUFFER+1(R5) ; Add 256. to buffer pointer.
      ;ADD #256.,BUFFER(R5) ; (above equivalent to this)
      BNE 10$ ; Br if no overflow of extended address
      ADD #10000,EXTADR(R5) ; Update extended address bits
10$:  MOV #WW,NEXTFUNC(R5) ; Next function is a write
      RETURN
.sbttl WW - Write Silo to Disk Action Routine
;+
;
; Write write - This routine writes the silo to the floppy.
; It tests to see if the transfer is complete, and if it is then
; the next function to be executed is initiated. The next function
; is either a zero fill (if required) or just a return. If the
; transfer is not finished, then it calculates the next device
; address to be used.
;
```

## SAMPLE DEVICE HANDLERS

```

;      R5 -> Impure area
;      R3 -> RX#2 CSR
;      R2 = RX#2 function code
;      R0 = Either word count or sector
;      R1 = Either buffer address or cylinder
;
;      CALL    WW
;
;-
;      R5 -> Impure area
;      R3 -> RX#2 CSR
;      R2 = RX#2 function code
;      R0 = Either word count or sector
;      R1 = Either buffer address or cylinder
;
;      CALL    WW
;
;-
;WW:   MOV     DONE(R5),NEXTFUNC(R5)      ; Assume we are done
;      CALL    WX                         ; Write silo to floppy
;      TST     BYTES(R5)                  ; See if we are done
;      BEQ    10$                        ; If EQ, yes, return
;      MOV     #WF,NEXTFUNC(R5)          ; Else, next is a fill
10$:   CALL    DADINC                   ; Calculate next disk address
      RETURN
.sbtll RE - Empty Silo to Buffer Action Routine
;+
;
; Read Empty - This routine dumps the silo to the user buffer.
; It also checks to see if transfer is complete
; and increments the buffer pointer and device address if it is not.
;
;      R5      -> Impure area
;      R3      -> RX#2 CSR
;
;      CALL    RE
;
;      R0,R1,R2               Modified
;
;-
RE:    CALL    WORDCT
      MOV     RETRYS(R5),RTYCNT(R5)      ; Determine # words to read
                                         ; Reset retry count in case there
                                         ; has been a corrected soft error
      MOV     #CSEBUF!CSGO!CSINT,R2      ; Tell 'execute' its an empty function
      MOV     BUFFER(R5),R1              ; Give 'execute' the buffer address
      CALL    EXECUT                   ; Execute the function
      CLR     NEXTFUNC(R5)             ; Set up to execute 'done' next
      TST     BYTES(R5)                ; Is word count going to be exhausted?
      BEQ    20$                      ; If EQ, yes, return, else
      INCB   BUFFER+1(R5)             ; Increment buffer pointer.
      ; ADD   #256.,BUFFER(R5)        ; (above effectively adds ?256.)
      BNE    10$                      ; Br if no overflow of extended address
      ADD    #10000,EXTADR(R5)         ; Update extended address bits
10$:   CALL    DADINC                   ; Calculate next disk address
      MOV     #RR,NEXTFUNC(R5)          ; Next function is read
20$:   RETURN
;
.sbtll RR - Read Sector to Silo Action Routine
;+
;
; Read Read - This action routine reads from the floppy to the silo.
;
;      R5      -> Impure area
;      R3      -> RX#2 CSR
;
;      CALL    RR
;
;      R0,R1,R2               Modified
;
;-

```

## SAMPLE DEVICE HANDLERS

```

RR:    MOV     SECTOR(R5),R0      ; R0 = sector
       MOV     TRACK(R5),R1      ; R1 = track
       MOV     #RE,NEXTFUNC(R5)   ; Next function is a read empty
       MOV     #CSRDI!CSGO!CSINT,R2 ; R2 = current function (READ)
       CALL    EXECUT            ; Execute this function
       CLR     LSTBYT(R5)        ; If an error for this function,
                                ; bytes does not need to be corrected
                                ; since we have not altered it yet.

       RETURN

.sbtll ZF - Zero Fill the Silo Action Routine
;+
;
; Zero Fill - This action routine writes a sector full of zeros,
; to the silo.
;
;      R5      ->          Impure area
;      R3      ->          RX#2 CSR
;
;      CALL    ZF
;
;      R0,R1,R2           Modified
;
;-
ZF:    MOV     #CSFBUF!CSGO!CSINT,R2 ; Writing a zeroed silo.
       MOV     #1,R0             ; Writes all zeros to the silo.
       MOV     R5,R1             ; Status is zero at this point.
       ADD     #STATUS,R1        ; R1 -> status word
       MOV     #ZW,NEXTFUNC(R5)   ; Next function is a write zeros.
       CALLR   EXECUT            ; Execute this function and return

.sbtll ZW - Write a Zeroed Silo Action Routine
;+
;
; Write Zeroes - This action routine writes a zero-filled silo
; to the disk.
;
;      R5      ->          Impure area
;      R3      ->          RX#2 CSR
;
;      CALL    ZW
;
;      R0,R1,R2           Modified
;
;-
ZW:    CLR     NEXTFUNC(R5)        ; Clear nextfunc tells dyint to stop
       .BR     WX              ; Go write it.

.sbtll Subroutines for Action Routines:
; The following are some of the service routines used by the above
; action routines:

.sbtll WX - Write Silo Subroutine
;+
;
; Write X - This routine writes the silo to the floppy. It is used by
; WW and by ZW, the routine which zeros the unused portion of a block.
;
;      R0,R1,R2           Modified
;
WX:    MOV     #CSWRT!CSGO!CSINT,R2 ; R2 = Command to execute
       MOV     SECTOR(R5),R0      ; R0 = Sector
       MOV     TRACK(R5),R1      ; R1 = Track
       CALLR  EXECUT            ; Execute the command and return

```

## SAMPLE DEVICE HANDLERS

```
.sbttl WORDCT - Set Up Byte Count Subroutine
;+
;
; WORDCT - Called to determine the number of words to be transferred during
; the next silo operation. The word count is either the next full sector
; or the remaining partial sector count.
;
;     CALL    WORDCT
;
;     R0      =          Number of words to be transferred
;     BYTES(R5)           adjusted by amount in R0
;
;-
;
WORDCT: MOV    #SECSIZ,R0      ; Sector size is usual number of bytes xfered
        CMP    R0,BYTES(R5)   ; Compare to remaining count
        BLOS   10$           ; If LOS, do full sector transfer
        MOV    BYTES(R5),R0   ; Else do partial sector transfer
10$:   SUB    R0,BYTES(R5)   ; Current byte count after xfer
        MOV    R0,LSTBYT(R5) ; Save number xfered for error correction.
        ASR    R0             ; Change bytes to words for RX02
        RETURN
.sbttl EXECUT - Execute a Controller Command Subroutine
;+
;
; EXECUT - This routine actually initiates the RXV21 function.
;
;     R5      ->          Impure area
;     UNIT(R5)           Unit number
;     R3      ->          Control and Status register (CSR)
;     R2      =            First argument of command
;     R1      =            Third argument of command
;     R0      =            Second argument of command
;
;     CALL    EXECUT
;
;     R2 has unit number and extended address bits set into command.
;
;-
;
EXECUT: BIS    UNIT(R5),R2      ; include unit bits
        BIS    EXTADR(R5),R2   ; and extended address bits
10$:   BITB   #CSDONE,@R3      ; Wait until RX02 is ready.
        BEQ    10$           ; If EQ, loop until it is
        MOV    R2,@R3           ; Send command.
20$:   BITB   #CSDONE!CSTRAN,@R3 ; Ready yet ?
        BEQ    20$           ; If EQ, loop until it is
        MOVB   R0,@RX2DB(R5)   ; Send next argument.
30$:   BITB   #CSTRAN!CSDONE,@R3 ; Wait till ready.
        BEQ    30$           ; If EQ, loop until ready
        MOV    R1,@RX2DB(R5)   ; Send third argument.
        RETURN              ; RX02 interrupts when done
;
.sbttl EXEC2 - Execute Two Argument Command Subroutine
;+
;
; EXEC2 - Executes the 2-argument RX02 commands (Read Maintenance
; and Format). This routine is run with interrupts (from the RX02)
; disabled, so it waits until the function is done.
;
;     R3      ->          Control and Status register (CSR)
;     R2      =            First argument of command
;     R0      =            Second argument of command
;
;     CALL    EXEC2
;
;-
```

## SAMPLE DEVICE HANDLERS

```

EXEC2: BITB #CSDONE,@R3 ; Is RX02 ready ?
       BEQ EXEC2 ; If EQ, loop until ready
       MOV R2,@R3 ; Send command.
10$:  BITB #CSDONE!CSTRAN,@R3 ; Wait till ready.
       BEQ 10$ ; If EQ, loop until ready
       MOV R0,@RX2DB(R5) ; Send second argument.
20$:  BITB #CSDONE,@R3 ; Wait till done.
       BEQ 20$ ; If EQ, loop until ready
       RETURN

.sbtll DAD - Device Address Calculation Subroutine
;+
;

; DAD - This routine does RTII-style 2:1 sector interleave with
; a six sector per track offset.

;
; R4      ->           Message packet
; R5      ->           Impure area
;
; CALL    DAD
;
; SECTOR(R5),TRACK(R5),BOUND(R5),LASTSC(R5) are updated.
; R0,R1 modified.
;
;-
.enabl LSB
pure$


DAD:
push$ <R2,R3> ; Save registers
TSTB ADTYPE(R5) ; Physical addressing?
BNE LOGIC ; If NE, no, logical address
MOV DP.CYL(R4),TRACK(R5) ; Yes, use given track(i.e., cylinder)
MOVB DP.SEC(R4),SECTOR(R5) ; and sector.
BR 70$ ; Return

; If logical we do the interleave calculation.

LOGIC: MOV DP.DAD(R4),R2 ; Make logical block into
       CMP #LBSIZE,R2 ; Is this a disk address overflow?
       BLO 70$ ; Br if address is not legal with
                ; carry set to mark error
                ; and return to caller.

;

; Divide by 26 and mod 26
;

; This section divides R2 by 26. and puts the remainder in R2 and the
; quotient in R1.

10$: ASL R2 ; Logical sector
       MOV #8.,R3 ; Set up shift counter.
20$: CMP #26.*200,R2 ;
       BHI 30$ ;
       ADD #-26.*200,R2 ; If yes subtract 26*2**8
30$: ROL R2 ;
       DEC R3 ;
       BGT 20$ ; Count number of shifts.
       MOVB R2,R1 ; Repeat till done.
       CLRB R2 ; Put the logical sector div 26 in R1
       SWAB R2 ; And the logical sector mod 26 in R2.
       MOV R1,TRACK(R5) ; Save track number.
       INC TRACK(R5) ; Track 0 is reserved, so inc track.
       CMP #12.,R2 ; Add 1 to sector if in 2nd half
       ROL R2 ; of track and multiply by 2.

```

## SAMPLE DEVICE HANDLERS

```

; Add offset ( track -1 * 6 )
        ASL      R1                      ; Multiply track by six.
        MOV      R1,R0
        ASL      R1
        ADD      R0,R1
40$:   SUB      #26.,R1                  ; Take mod 26 of offset.
        BGE      40$ 
        ADD      #26.,R1

50$:   MOV      R1,BOUND(R5)            ; This is where sec go from odd to even
        INC      BOUND(R5)
        ADD      R1,R2                  ; We save it for DADINC.
        TST      R1
        BNE      60$                  ; If sector is zero, we want 26.
        MOV      #26.,R1
60$:   SUB      #26.,R2                  ; Find sector mod 26
        BGE      60$ 
        ADD      #27.,R2
        MOV      R2,SECTOR(R5)          ; and save
        MOV      R1,LASTSC(R5)          ; This is the last sector on the track
        CLC
70$:   pop$    <R3,R2>                ; Clear carry - no error
        RETURN
        .dsabl LSB
.sbtll REPLY - Return Status Message Subroutine
;+
;
; This routine dispatches the reply to the user if he requested one.
;
;      R4      ->      Message packet
;      R5      ->      Impure area
;
; Output:
;
;      R4      =      0  (Packet no longer available)
;
;-
;
        .enabl LSB
pure$ 

REPLY:
        TST      R4                      ; R4 -> Request packet
        BEQ      80$                  ; If EQ, none
10$:   MOV      R1,-(SP)                ; Save register
        MOV      R4,R1                  ; -> sdb of the reply sem.
        ADD      #DP.SEM,R1
        TST      @R1                   ; Did he specify a reply semaphore ?
        BEQ      20$                  ; Br if not
        BIT      #FMSBSM,DP.FUN(R4)  ; Did he specify a binary semaphore ?
        BEQ      50$                  ; Br if not, send a full reply
        sgnl$  R1                     ; No, just signal the binary semaphore

;***** Check for error on signal
        BR      20$                  ; Return queue element to the pool

; Full reply

; Put status, error code, and actual byte count in the appropriate places
; in the reply queue element.

```

## SAMPLE DEVICE HANDLERS

```

50$:    MOV      DP.LEN(R4),DP.ALN(R4) ; Copy byte count requested
        MOV      STATUS(R5),DP.STS(R4) ; Copy status code
        BEQ      60$                 ; If EQ, all bytes were xfered.
        assum$  IS$NOR EQ 0
        ADD      LSTBYT(R5),BYTES(R5) ; Xfer did not complete so reset byte
        SUB      BYTES(R5),DP.ALN(R4) ; Subtract number left from total .
        60$:    MOV      ERROR(R5),DP.ERR(R4) ; Put error code from READM in queue.
        MOVB   #DP.SIZ-SE.HDR,SE.CTL(R4) ; Put packet size in control byte
        sglq$  R1,R4                ; Send the reply.
        BCC    70$                 ; Br if no error
        20$:    dapk$  R4             ; Return queue element to the pool
        70$:    CLR      R4             ; No more access to that queue element
        MOV      (SP)+,R1            ; Restore register
        80$:    RETURN               ; and return

        .dsabl LSB
.sbtll DADINC - Increment Device Address Subroutine
;+
;
; DADINC - This procedure does an interleaved or physical increment of the
; sector and track for the RX02 .
;
;      R5 -> Impure area
;
;      CALL DADINC
;
;      SECTOR,TRACK,BOUND,LASTSC,ADTYPE are updated.
;      R0 modified.
;
;-
;
.enabl LSB

DADINC:
        push$  R2                  ; Save register
        TSTB   ADTYPE(R5)          ; Test addressing mode
        BEQ    PHYS                ; If EQ, physical address
        BLT    FIRST               ; If LT, first sector on a track
                                ; requires special handling

; Increment a logical address:

        MOV      SECTOR(R5),R2       ; Add 2 to sector
        ADD      #2,R2
        CMP      #26.,R2
        BGE    10$
        SUB      #26.,R2
        10$:   CMP      R2,LASTSC(R5) ; If it's the last sector on the
        BNE    20$                 ; track, mark next sector as
                                ; a new track
        NEGB   ADTYPE(R5)
        20$:   CMP      R2,BOUND(R5) ; See if it's time to go even
        BNE    30$                 ; Map into the even sectors
        INC      R2
        30$:   MOV      R2,SECTOR(R5)
        BR     70$                 ; Return

; Calculate the starting sector of a new track:

FIRST:  NEG B   ADTYPE(R5)          ; Clear new track flag
        MOV      TRACK(R5),R2        ; Calculate new track offset
        ASL      R2
        MOV      R2,R0
        ASL      R2
        ADD      R0,R2              ; Multiply track by 6
                                ; to get offset

40$:    SUB      #26.,R2          ; Last sector is offset
        BGT    40$                 ; mod 26
        ADD      #26.,R2            ; If mod 26 = 0, LASTSC = 26

```

## SAMPLE DEVICE HANDLERS

```
50$:    MOV     R2, LASTSC(R5)          ; This is the last sect written  
        TST     R2  
        BNE     60$  
        MOV     #26., LASTSC(R5)  
  
60$:    INC     R2                  ; Add 1 to get first sector  
        MOV     R2, BOUND(R5)          ; This address marks the  
        MOV     R2, SECTOR(R5)          ; odd/even boundary  
        INC     TRACK(R5)            ; First sector on track  
        CMP     TRACK(R5), #NUMCYL   ; New track  
        BLE     70$  
        MOV     #ISSIDA, STATUS(R5)  ; Is track legal ?  
        BR      70$                ; If LE, yes  
                                ; No, return invalid device address  
                                ; Return  
  
PHYS:   INC     SECTOR(R5)          ; For physical reads writes  
        CMP     #NUMSEC, SECTOR(R5)  
        BGE     70$  
        MOV     #1, SECTOR(R5)          ; Increment sector  
        INC     TRACK(R5)            ; If GE, sector within bounds  
        CMP     #1, SECTOR(R5)          ; Else sector overflow, reset  
        BR      70$                ; And increment track number  
  
70$:    pop$    R2  
        RETURN  
  
.dsabl LSB  
.end
```

## SAMPLE DEVICE HANDLERS

### A.2 DLV11 (XL) HANDLER--SAMPLE MACRO-11 DEVICE-HANDLER SOURCE

```
.nlist          ;Edit Level 0
.enabl  LC
.list

.title  XLDRV  Serial Line (XL) Device Driver
.ident  /V01.00/

;
;           COPYRIGHT (c) 1982 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.
;

;+
;

; Module name:  XLDRV.MAC
;

; System: MicroPower/Pascal RTS
;

; Author: KTP          Creation date: 14 Oct 88
;

; Modified:
;

;

; Functional Description:
;

;     This module provides serial line device driver services.
;     It controls any number of DLV-11 type serial lines, and
;     provides the following functions:
;

;         o Block mode reads and writes
;         o Connect and disconnect ring buffers for receive or transmit
;         o Get and set status for modem control and programmable baud rate
;         o Wait for modem change of state
;

;+
;

; Assembly instructions:
;

; MACRO XLDRV,XLDRV=XLDRV,COMU,SML/M
;-
; .sbttl Declarations

;+
;

; System macro requires
;-
```

## SAMPLE DEVICE HANDLERS

```
.mcall MACDFS,IODFS,QUEDFS,DRVDFS,XLISZS,MISDFS  
  
macdfs  
iodfs  
quedfs  
drvdfs  
misdfs  
xlisz$  
  
.mcall ALPCSS,CRSTS,CRSTSS,CRPC$,DLSTS,DLPC$,STPC$,ALPK$,DAPK$  
.mcall CINT$,WAIQ$,SGLQS,SGLQSS,GELMS,WAIT$,CHGP$,CRPC$  
.mcall WAIT$,SGNL$,SGNL$S,GELO$,PELC$,DFSPCS,FORK$,CRPC$P  
.mcall DRMAP$,MVBYTS,IBADRS  
  
dmap$ p1=$XLADDR,pf=XLSPR,*vp=PT,DRF,IRFL,IRFL,IRFL,IRFL,IRFL,IRFL,IRFL  
;  
; Local macro definitions  
;  
 .macro HALTS  
 BR  
 .endm  
;  
; All external symbols which are necessary to assemble this module but  
; which may have default definitions:  
;  
 dfalt$ BS.IZE,16. ; Input burst mode buffer size  
 dfalt$ IO.BS2,16. ; Output burst mode buffer size  
;  
; The following symbols are input burst mode buffer symbols  
;  
 orig$ .TEMP.,<RO,D,LCL,ABS,OVR>  
  
.temp.:  
B.BUF$ .blkw ; -> Next free element in buffer  
B.CCNT: .blkw ; Number of characters in buffer  
B.ABP: .blkw ; -> Alternate buffer  
B.BUFF: .blkb BS.IZE ; Buffer  
.even  
BE.SIZ = . - .temp. ; Size of buffer control block  
;  
; Offsets into ISR Impure area.  
;  
; Each line has an impure area associated with it. All the impure areas  
; are linked together so that the control process can check each line for  
; action requests.  
;  
dorg$ DLIMP,$,GLOBAL  
  
dword$ I.FLAG ; Flag Word. Individual bits defined below  
dword$ I.ERR ; Error Word. Individual bits defined below  
dword$ I.RCSR ; Saved contents of RCSR  
dword$ I.XFLG ; Pointer to Flag Word of transmitters impure area  
 ; (only used in input side)  
dword$ I.HBUF ; Address of hardware RBUF/XBUF  
dword$ I.CHAR ; Current character being output  
dword$ I.BUF ; Saved contents of RBUF  
dword$ I.PAR ; PAR to users data block  
dword$ I.ADR ; Virtual address of user's data block
```

## SAMPLE DEVICE HANDLERS

```
; Arg block for PELM/GELM in ring buffer mode

dword$ I.RBF           ; Pointer to ring buffer SDB
dword$ I.DAT           ; Pointer to I.CHAR
dword$ I.CNT           ; Byte count, set to 1 char
dword$ I.RDB,3          ; SDB for ring buffer

dword$ I.RQUE          ; Queue of user request packets
dword$ I.LQLM           ; Holds current user request packet
dword$ I.LINK           ; Pointer to impure area of next Serial Line unit
dword$ I.CTS.:          ; Pointer to following semaphore SDB for SGNL
dword$ I.CSDB,3          ; SDB for control process semaphore

; The following section is only present for the receiver side of a line

dbyte$ I.IBFL,BE.SIZ   ; First burst mode buffer
.deven
dbyte$ I.IBF2,BE.SIZ   ; Second burst mode buffer
.even
dword$ I.IBCB           ; -> ISR burst mode buffer control block
dword$ I.FRKS           ; 0 if fork in progress, 1 if not
; *** Adjacency required for the following 2 bytes
dbyte$ I.CLAS           ; Device class as specified in the prefix file,
; should be DC$TER
dbyte$ I.TYPE           ; Device type, i.e. TT$DL, TT$DLE, TT$DLF, etc.
dword$ I.SQUE           ; Report change in status queue pointer
.even
dsizes$ ISIZ.I          ; Impure allocation for receiver

; The following section is only present for the transmitter side of a line

.      =      I.IBFL ; Overlaps unique section of input size

dword$ I.HLSM           ; Pointer to following semaphore SDB for SGNL
dword$ I.HSDB,3          ; SDB for helper process semaphore
dword$ I.HSNM           ; Null name for above

dword$ I.HPDB,3          ; PDB for helper ISR process
dword$ I.HPNM           ; Null name for above
dword$ I.XBP             ; -> Output burst mode buffer
dbyte$ I.XBUF,IO.BSZ    ; Output burst mode buffer
dsizes$ ISIZ.O          ; Impure allocation for transmitter

HST.SZ == $MINSTK + 20. ; Initial allocation for helper stack
dword$ I.GRD1           ; Space for guard word
dbyte$ I.HSTK,HST.SZ    ; Stack for per-line ring buffer helper process
dword$ I.GRD2           ; Space for guard word

dsizes$ I.SIZ           ; Size of total impure area excluding -2 word

.if LT <XL$IIISZ - ISIZ.I>
    .error XL$IIISZ ; Configured input impure area too small
.endc
.if LT <XL$OISZ - ISIZ.O>
    .error XL$OISZ ; Configured output impure area too small
.endc
.if LT <XL$SSZ - HST.SZ>
    .error XL$SSZ ; Configured stack area too small
.endc
    .sbttl Definitions of bits in flag word I.FLAG
;+
; Definition of Bits in main control flag word, I.FLAG
;-
```

## SAMPLE DEVICE HANDLERS

```
borg$ FLGBT,0,GLOBAL
bit$ FSATTN          ; This line needs attention of control process
bit$ F$DONE           ; Block transfer is complete
bit$ F$DSAT           ; Data Set wants attention
bit$
bit$ F$IDLE           ; Line is idle
F$DSAC::             ; Data Set is active. (used on input side)
bit$ F$INHO           ; Inhibit output (used on output side only)
bit$ F$CTRL           ; Controller detected error
bit$

bit$ 
bit$ 
bit$ 
bit$ F$XCHK           ; Check for and process X-OFF and X-ON
; *** NOTE *** F$XCHK must = FM$CHK
bit$ F$RING           ; Indicates that ring buffer is connected to line
bit$ F$CNFG           ; Indicates ring buffer created at configuration time
bit$ 
bit$ 

        .sbttl Definition of bits in error word I.ERR
;+
; Definition of hardware/software error bits in error word, I.ERR
;-

borg$ ERRBIT,0,GLOBAL
bit$ 
bit$ 
bit$ 
bit$ 
bit$ 
bit$ 
bit$ E$XFOV            ; Transmitter FORK overrun
bit$ E$BOVR           ; Buffer overrun. Ring buffer has no room
bit$ E$FOVR            ; FORK overrun, second interrupt before first is done
bit$ 
bit$ 
bit$ 
bit$ 
bit$ E$DPTY            ; Device parity error from RCSR
bit$ E$DFRM            ; Device framing error from RCSR
bit$ E$DOVR            ; Device overrun from RCSR
bit$ E$ERR              ; This bit is set when any error occurs

E$NONE == C<E$ERR ! E$DOVR ! E$DFRM ! E$DPTY>; Mask for clearing non-error
; bits in RCSR

FNCOD = 77             ; Function code takes up 6 bits, 0-5
MAXCMD = 14             ; Maximum function code is 14
        .sbttl CSR and BUF register definitions

;+
; Control and Status, Buffer register layouts
;-

;
; Receiver CSR
;
```

## SAMPLE DEVICE HANDLERS

## SAMPLE DEVICE HANDLERS

```
; Transmitter buffer
;

borg$    XBUF,0,GLOBAL
bit$          ; Receive data
bit$          ;
bit$          ; unused
        .sbttl Miscellaneous definitions
;+
; Miscellaneous definitions
;-

;+
; Characters for X-OFF/X-ON checking
;-

XOPCHR = 'S-100      ; ASCII X-OFF
XONCHR = 'Q-100      ; ASCII X-ON

;+
; Hardware register offsets relative to RBUF. ISR is entered with pointer
; to RBUF.
;-
CSR      = -2          ; CSR is, in general, 2 before BUF.

dorg$   REGSET,0,GLOBAL
.       =     . - 2
dword$  RCSR          ; RCSR is 2 before RBUF
dword$  RBUF          ; ISRs entered with R4 pointing to hardware BUF.
dword$  XCSR          ; XCSR is 2 after RBUF. XCSR accessed from input ISR
                      ; when processing X-ON.
dword$  XBUF

; Miscellaneous
SIDLEN = 3           ; Structure ID length (words)
PIDLEN = 3           ; Process ID length (words)
        .sbttl Impure (R/W) data
        impur$

; SDB for XL Driver user request semaphore.

$XLAQD:::blkw  SIDLEN      ; Space for structure identifier
        .ascii  /$XLA /      ; Semaphore name

; PDB for Driver process.

DLPDDB: .blkw  PIDLEN      ; Process identifier
        .ascii  /      /      ; Process name (filled in by init routine)
```

## SAMPLE DEVICE HANDLERS

```
; DL Control process stack.

DLCSTT: .word 125252          ; Top of stack
        .blkw XL$SSZ
DLCSTK: .blkw                 ; Stack starting location and guard word

; PDB for Control process.

DLCPDB: .blkw PIDLEN         ; Process identifier
        .ascii / /           ; Process name (filled in by init routine)

; DL Control process synchronizing binary semaphore SDR.

DLCSEM: .blkw SIDLEN         ; Semaphore identifier
        .word 0              ; Null name

;+
; Each serial line has an impure area for both the input side and the
; output side. Each area is linked to the next line on the same side
; and the output chain is linked to the end of the input chain. The
; words below point to the start of each chain. During initialization
; this is indexed as a two word table.
;-
DLIBP: .word 0                ; Input line data block chain pointer
DLOBP: .word 0                ; Output line data block chain pointer

; The variables below are used during line initialization.

COUNT: .word 0                ; Running count of lines
CHNPNT: .word 0               ; Running pointer to last link in chain on
        .word 0               ; input and output side. These two words
        .word 0               ; are indexed as a two word table

; Locations used for predeclared ring buffers; adjacency assumed.

CONUNI: .BLKW                ; Unit number we are preconnecting
RSDBPT: .BLKW                ; Pointer to ring buffer SDB below
RNGTYP: .BLKW                ; Contains ST.RBF
RNGATT: .BLKW                ; Gets attributes passed in prefix file
RNGSIZ: .BLKW                ; Gets size of buffer declared in prefix
RNGSDB: .BLKW 6              ; SDB for preconnected ring
RNGFUN: .BLKW                ; Function modifiers (such as X-ON checking)

; Static process stack

STKTOP: .blkb XL$SSZ         ; Allocate space for stack
STKINI:                      ; Initial stack value same as bottom value
STKBOT: .blkw                 ; Bottom of stack and guard word
        .sbttl Pure (R/O) data
        pdat$

;+
; Initialization data - table of ISR entry points
;-
ISRPNT: .word $IISLD          ; Table of interrupt service routine entries
        .word $OISLD          ; indexed by R5 in cint$
```

## SAMPLE DEVICE HANDLERS

## SAMPLE DEVICE HANDLERS

```

        MOV      R2,I.HBUF(R3)    ; Save hardware BUF address
        MOV      (SP)+,R0          ; Restore configuration table pointer
        RETURN

;+
;CONRNG
;
; Routine to create and connect predeclared ring buffers defined
; in the prefix file.
;
; R0      ->      Ring buffer name in the configuration table
; R2      ->      Impure area
;
; JSR     R5,CONRNG
; .word   IFSCRR or IPSCXR for receive or transmit respectively
;-

CONRNG: MOV      (R0)+,RNGSDB+6 ; Copy first word of name if any
            BEQ      3WS      ; Branch out if no ring declared
            assum$ I.FLAG EQ #0
            BIS      #FSCONF,(R2) ; Indicate line connected to ring buffer
                                ; at configuration time
            MOV      (R0)+,RNGSDB+14 ; Else copy rest of name
            MOV      (R0)+,RNGSDB+12 ;
            CLR      RNGSDB       ; Clear any old id
            MOV      (R0)+,RNGSIZ  ; Pick up size of the ring
            MOV      (R0)+,RNGATT  ; Pick up attributes
            MOV      (R0)+,RNGFUN  ; and get function modifiers
            MOV      R0,-(SP)      ; Save R0 across CRST
            MOV      #RSDBPT,R0    ; Point at CRST parameter block
            MOV      #RNGSDB,(R0)  ; Set SDR pointer in it
            MOV      #ST.RBF,RNSTYP ; Set type of structure to ring
            crsts$ Create the ring
            BCC      1WS      ; Br if no error
            CALL    SDEGCS       ; Report error (Does not return)

1AS:  alpkcs RI      ; Get a packet
            MOV      (R5)+,DP.FUN(R1) ; Set function code
            BIS      RNGFUN,DP.FUN(R1) ; Or in function modifier bits
            MOV      RNGSDB,DP.SEM(R1) ; Set SDB for ring we just created
            MOV      RNGSDB+2,DP.SEM+2(R1)
            MOV      RNGSDB+4,DP.SEM+4(R1)
            MOV      CONUNI,DP.UNI(R1) ; Set unit number
            sqlqss $SKLAOD,RI      ; Send ourselves a connect request
            BCC      2WS      ; Br if no error
            CALL    SDESID       ; Report error (Does not return)

20S:  MOV      (SP)+,RA      ; Restore RA
            RTS      R5
30S:  TST      (R5)+      ; Link the parameter passed in-line
            RTS      R5
            .sbttl  KL Driver Initialization process

;+
; The DINIT process runs at system initialization priority and performs
; only those functions needed to permit requests to be entered without
; error. The main request (-Semaphore is created and the main driver
; process is created without a name. Then the DINIT process is deleted.
; Actual line initialization is done when the handler process is run which
; may not happen until later depending on the relative priority of other
; processes.
;-

```

## SAMPLE DEVICE HANDLERS

```

$XL::: MOV      #"$X,$XLAQD+SD.NAM      ; Set name of request Q semaphore
      MOV      #"$LA,$XLAQD+SD.NAM+2
      MOV      #"$X,$XLAQD+SD.NAM+4
      CLR      $XLAQD                      ; Clear possible old id in case of restart
; Create the request Q-Semaphore

      crst$$s sdb= $XLAQD,styp= ST.QSM,satr= SA$IPR!SA$OPR,value= 0
      BCC      10$                      ; Br if no error
      CALL     $DEGCS                   ; Report error (Does not return)

      .sbttl XLDdrv data initialization
;+
; DLDdrv is the main DL request processor. When first run it performs some
; data initialization, then it initializes each serial line. After this
; initialization, it enters the main request processing loop, and waits
; for a user request.
;-
; Create the control semaphore: the control process waits on this semaphore,
; which is signalled by the ISRs whenever some action must be performed by
; the control process.

10$:   CLR      DLCSEM                  ; Clear id field in case of restart
      crst$$s sdb= DLCSEM,styp= ST.BSM,satr= SA$OPR,value= 0
      BCC      20$                      ; Br if no error
      CALL     $DEGCS                   ; Report exception (Does not return)

; Create control process

20$:   CLR      DLCPDB                  ; Clear id field in case of restart
      MOV      #"$X,DLCPDB+SD.NAM+2    ; Init process name field
      MOV      #"$LC,DLCPDB+SD.NAM+4
      MOV      #"$TL,DLCPDB+SD.NAM+4
      crpc$  area=CTRLPD                ; Create control process
      BCC      30$                      ; Br if no error
      CALL     $DEGCP                   ; Report exception (Does not return)
; Initialize line data block chain pointers.

30$:   MOV      #DLIBP,CHNPNT          ; Input chain start
      MOV      #DLORP,CHNPNT+2         ; Output chain start
      MOV      #SXLPRM,R0              ; Start of DL configuration table
      MOV      (R0)+,COUNT            ; Initialize line count and point to
                                      ; first vector value

      .sbttl XLDdrv line initialization

; Start of line initialization loop. Re-enter here for each new line.

40$:   CLR      CONUNI                 ; Clear unit for ring preconnects
      CLR      R5                     ; R5 serves as control and index
                                      ; between input and output side
                                      ; of each line
      MOV      (R0)+,R1              ; Pick up vector address for this line
                                      ; Will be incremented by 4 for output
                                      ; side of same line.
      MOV      (R0)+,R2              ; Pick up RCSR. R2 will scan thru
                                      ; all 4 device addresses
      MOV      (R0)+,R3              ; -> Input ISR impure area
      assum$ <I.CLAS1> EQ 0        ; Start on word boundary?
      assum$ I.TYPE EQ <I.CLAS+1>  ; Adjacency?
      MOV      (R0)+,I.CLAS(R3)       ; Device class and type

```

## SAMPLE DEVICE HANDLERS

```

CLR    I.SQUE(R3)           ; Clear FIFO queue of requests for
                            ; reports of status changes
CALL   SETUP                ; Call routine to initialize the line
                            ; data area (input line)
ADD    #2,R5                ; Index to output side
MOV    R3,-(SP)             ; Save address of input line data area
ADD    #4,R1                ; R1 -> output vector
assum$ XCSR-RBUF EQ 2      ;
TST    (R2)+                ; R2 -> XCSR
MOV    (R0)+,R3              ; -> Output ISR impure area
CALL   SETUP                ; Call routine to initialize the line
                            ; data area for the output line
MOV    (SP),R2              ; Get base of input side of data area
assum$ I.FLAG EQ 0          ;
MOV    R3,I.XFLG(R2)         ; Save pointer to output side flag
                            ; word in input side data area
                            ; (for X-ON/X-OFF)
JSR    R5,CONRNG            ; Connect possible receive ring
.word  IF$CRR
MOV    R3,R2                ; -> Output ISR impure area
JSR    R5,CONRNG            ; Connect possible transmit ring
.word  IF$CXR
MOV    (SP)+,R5              ; -> Input ISR impure area
INC    CONUNI               ; Up the unit counter
DEC    COUNT                ; Count the line
BGT    40$                  ; If not 0 go do another
CLR    I.LINK(R3)            ; Clear link word of last impure
;+
; Lines all set up. Set last input chain pointer to point to first output
; line. Control process uses this chain to poll all lines starting with the
; input lines then proceeding directly to output lines.
;-
MOV    DLOBP,I.LINK(R5)      ;
chgp$  #XLSPPR+1            ; Now drop our priority to running level
.BR    $XLREQ                ; Enter main request processing loop
                            ; of driver
.sbtcl Command Dispatching
.enabl lsb

;+
; This section waits for requests from users. READ and WRITE requests are
; dispatched to the proper sub-Q for the unit requested. GET and SET requests
; are processed immediately.
;-
$XLREQ::
waiq$  #SXLAQD,R1           ; Wait for a request
BCC    10$                  ; Br if no error
CALL   $DESID               ; Report exception (Does not return)
10$:   MOV    R1,R5             ; Make R5 point to reply semaphore SDB, if any
       ADD    #DP.SEM,R5          ; (Q element base + offset)
;+
; Locate the line data blocks for the proper DL. R2 will point to input data
; block, R3 to the output block. R0, with the unit number, is used to count
; the number of steps thru the data block chain.
;-
MOV    DLIBP,R2              ; Points to first input block for Unit #
MOV    DLOBP,R3              ; Points to Unit 0 output block
MOVB   DP.UNI(R1),R0          ; Pick up unit number
BEQ    30$                  ; Branch if unit 0

```

## SAMPLE DEVICE HANDLERS

```

20$:    MOV      ILINK(R2),R2 ; Get next pointer for input side
        MOV      ILINK(R3),R3 ; and for the output side
        BEQ      120$          ; Branch if no impure allocated, no unit
        DECB     R0              ; Down the unit number
        BNE      20$            ; Loop if not correct unit

; Pick up and dispatch command

30$:    MOV      DP.FUN(R1),R4 ; Get function code
        BIC      #^CFNCOD,R4 ; Clear command modifier bits
        CMP      R4,#MAXCMD   ; Range check command code
        BGE      130$          ; Branch if command code too big
        CMPB     R0,$XLPRM    ; Valid unit number?
        BGE      120$          ; Br if unit number non-existent
        ASL      R4              ; Double code to use as word index
        CALL     @CMDTBL(R4)   ; CASE (DP.FUN) OF
        BR      $XLREQ          ; Loop -- process another request

;+
; Commands are entered with:
;
; R0      = garbage
; R1      -> request packet
; R2      -> line input data area
; R3      -> line output data area
; R4      = function code * ?
; R5      -> reply semaphore SDR
;
; Command processing code should return via a RETURN.
;-

pdat$


assum$ MAXCMD EQ 14

assum$ IF$RDP EQ 0      ; Read physical
assum$ IFSRDL EQ 1      ; Read logical
assum$ IFSWTP EQ 3      ; Write physical
assum$ IF$WTL EQ 4      ; Write logical
assum$ IFSSET EQ 6      ; Set status
assum$ IFSGET EQ 7      ; Get status
assum$ IF$CRR EQ 10     ; Connect receiver ring buffer
assum$ IF$CXR EQ 11     ; Connect transmitter ring buffer
assum$ IF$DRR EQ 12     ; Disconnect receiver ring buffer
assum$ IF$DXR EQ 13     ; Disconnect transmitter ring buffer
assum$ IF$RSC EQ 14     ; Report status change

CMDTBL: .word 50$          ; Read physical
        .word 50$          ; Read logical (same as read physical)
        .word 130$         ; Read virtual (illegal command)
        .word 70$           ; Write physical
        .word 70$           ; Write logical (same as write physical)
        .word 130$         ; Write virtual (illegal command)
        .word 90$           ; SET status
        .word 110$          ; GET status
        .word 40$           ; Connect receiver ring buffer
        .word 60$           ; Connect transmitter ring buffer
        .word 150$          ; Disconnect receiver ring buffer
        .word 160$          ; Disconnect transmitter ring buffer
        .word 220$          ; Report status change

pure$               ; Back to code psect
.sbtll Command processing

;+
; Read and write commands simply insert the user's packet on a FIFO
; queue pointed to by the line's impure area and signal the control process.
; This queue is checked for entries when the control process looks for work.
;-

```

## SAMPLE DEVICE HANDLERS

```
; Read and connect receive ring commands

40$:    MOV      R2,R3          ; Copy input impure pointer
;       .BR      70$          ; Finish in write routine

; Write and connect transmitter ring commands

60$:    ADD      #I.RQUE,R3    ; Set pointer to request packet queue
CALL    ENQUE             ; Add request to the queue
sgn1$  #DLCSEM            ; Signal the control process
BCC    80$                ; Br if no error
CALL    SDESID             ; Report exception (Does not return)

80$:    RETURN

; SET Characteristics

90$:    MOV      DP.RPS(R1),-(SP) ; Copy receiver characteristics
BIS      #RCSINT,(SP)        ; Make sure interrupts are enabled
MOV      I.HBUF(R2),R4        ; -> RBUF
MOV      (SP)+,-(R4)         ; Load the RCSR
MOV      DP.XPS(R1),-(SP)    ; Copy transmitter characteristics
BIC      #XCSINT,(SP)        ; Clear the interrupt enable bit
MOV      I.HBUF(R3),R4        ; -> KBUF
SPL      XLSHPR              ; Disable DL-11 level interrupts
MOV      -(R4),R0              ; * Copy current XCSR interrupt enable setting
BIC      #^C<XCSINT>,R0      ; * Clear all but interrupt enable bit
BIS      R0,(SP)              ; * Copy current interrupt enable setting
MOV      (SP)+,(R4)           ; * Load the XCSR
MOV      DP.RSS(R1),R0        ; * Copy software status bits
BIC      #^C<FSXCHK>,R0      ; * Only interested in XON/XOFF checking bit
assum$  I.FLAG EQ 0
BIC      #FSXCHK,(R2)        ; * Clear XON/XOFF check bit in FLAG word
BIS      R0,(R2)              ; * Copy XON/XOFF check bit setting
TST      R0
BNE    190$                  ; * Did we just set FSXCHK?
assum$  I.FLAG EQ 1
BIT      #FSINHO,(R3)        ; * Output currently inhibited?
BEQ    190$                  ; * Br if not, no problem
BIC      #FSINHO,(R3)        ; * Enable output
BIS      #XCSINT,(R4)        ; * Enable output interrupts
100$:   SPL      R0
assum$  ISSNOR EQ 0
CLR      DP.STS(R1)          ; Send back success code
BR      190$                  ; -> RBUF-2> EQ RCSR

; GET Characteristics

110$:   MOV      I.HBUF(R2),R0    ; -> Receiver buffer
assum$ <RBUF-2> EQ RCSR
MOV      -(R3),DP.RPS(R1)    ; Return receiver physical state
MOV      XCSR-RCSR(R2),DP.XPS(R1) ; Return transmitter physical state
assum$ DP.TYP EQ <DP.CLS+1>; Adjacency?
assum$ I.TYPE EQ <I.CLAS+1>; Adjacency?
MOV      I.CLAS(R2),DP.CLS(R1) ; Return device class and type
assum$ I.FLAG EQ 0
MOV      (R2),DP.RSS(R1)      ; Return receiver software state
assum$ I.FLAG EQ 1
MOV      (R3),DP.XSS(R1)      ; Return transmitter software state
assum$ ISSNOR EQ 0
CLR      DP.STS(R1)          ; Send back success code
BR      190$                  ; -> RBUF-2> EQ RCSR
```

## SAMPLE DEVICE HANDLERS

```
; Set error codes
120$: MOV     #ISSNXU,DP.STS(R1)      ; Non-existent unit error
      BR     140$               

130$: MOV     #ISSFUN,DP.STS(R1)      ; Illegal function code
      BR     140$               

140$: CLR     DP.ERR(R1)           ; Clear extended error bits
      BR     180$           ; Now reply to user
;+
; Disconnect receive ring buffer
;-
150$: MOV     R2,R3           ; Copy impure pointer
      .BR    160$           ; Continue in transmit disconnect routine
;+
; Disconnect transmit ring buffer
;-
160$: BIT     #FSRING,(R3)        ; Test if ring buffer connected
      BEQ    130$           ; Illegal function if not
      BIC     #FSRING,(R3)        ; Clear ring mode bit
      BIS     #FSIDLE,(R3)        ; Set line idle, available for more work
      TST     I.HSDB(R3)         ; Test if helper process exists
      BEQ    170$           ; Branch if not, receive case
      stpc$  I.HPDB(R3)         ; Stop process if so, transmit case
170$: CLR     I.HSDB(R3)         ; Clear helper process semaphore
      sgnl$  #DLCSEM           ; Signal control process to allow any
                                ; possible waiting requests to be started now
      BCC    200$           ; Br if no error, return packet to pool and exit
      CALL   $DESID            ; Report exception (Does not return)

;+
; REPLY
;
; Notify user when done with request "immediate mole" requests
;
; Input:
;
;     R1      ->      Packet
;     R5      ->      Semaphore descriptor block
;
; Output:
;
;     R0          Modified
;-
;
; R5 was set on entry to point to reply semaphore SDR in request packet

REPLY:
180$: MOVB   SE.CTL(R1),R0      ; Get packet size
      BICB   #200,R0           ; Remove reference flag leaving value
                                ; count
      CMPB   R0,#DP.SEM+6-SE.UDF ; Can packet contain SDR?
      BLT    200$           ; Br if not
      TST    (R5)             ; Test if SDR passed
      BEQ    200$           ; Branch if none, just deallocate packet
      BIT     #FM$BSM,DP.FUN(R1) ; Test if binary or queue semaphore
      BEQ    190$           ; Branch if not binary
      sgnl$  R5               ; Signal the binary semaphore
                                ; Ignore any errors
      BR     200$           ; Deallocate packet and exit
190$: MOVB   #DP.SIZ-SE.UDF,SE.CTL(R1); Amount of value data returned and
                                ; no reference data
```

## SAMPLE DEVICE HANDLERS

```

        sqlq$S R5,R1          ; Signal the user's queue semaphore
        BCC  210$              ; Branch if successful, packet queue
200$: dapk$S R1            ; Return packet to kernel pool
210$: RETURN              ; All set, exit

;+
; Report change in status
;-
220$: MOV    R2,R3          ; -> Input ISR input area
      ADD    #I.SQUE,R3          ; -> Report status change queue
      CALL   ENQUE              ; Add packet to the queue
      alpc$S R3                ; Set a packet to reply with
      TST    R3                ; Packet available
      BNE   110$              ; Br if so, return success
      MOV    #ISSPNA,DP,TT(R2)  ; Resource famine, unable to perform
                                ; request.
      BR    180$              ; Notify user

.dsabl 1sb

;+
; ENQUE
;
; Append packet to a queue.
;
; Input:
;
;     R1      ->      Packet
;     R3      ->      Head of queue
;
; Output:
;
;     R3      Modified
;
;-
; lenab! 1sb
14$: MOV    R3,R1          ; Link current to next element
ENQUE:
      TST    R3                ; Test if end of list
      BNE   14$                ; If not, loop
      MOV    R1,R3              ; Else put packet in front of list
      CLR    R1                ; Clear current and to indicate end of list
      RETURN

.dsabl 1sb

;+
; Process termination info
;-

DLDTRM: stpc$S #DLCR9          ; Terminate current request
      l1p$C 1                ; Don't delete interrupt
      .spch! Internal C-Driver interrupt Processor

;+
; The internal control driver waits for an interrupt to be generated
; which is signalled by the Mail Handler process when it has put an
; Read or Write request. It is signalled by the interrupt service routine
; when any attention condition occurs internally, when the write I/O
; is started by any condition, or when all of the I/O is done. The input lines
; are checked before the output lines. The lines are sorted in the order
; in which they are given in the interrupt enable.
;
;-
; lenab! 1sb
XLCRTL:
      wait$S #DLCR9          ; Wait for interrupt from mailer or file
      R2$  14$                ; If interrupt
      DALE  14$,14$           ; Report error to mailer

```

## SAMPLE DEVICE HANDLERS

```

10$:    MOV      DLIBP,R3          ; Initialize pointer to character data blocks
        ; for each line. Output lines are linked
        ; after input lines.

20$:    TST      R3              ; Test if another impure block
        BEQ      $XLCTL           ; Enter wait state if not, this one is free

        .sbttl  XI Line Control
;+
; First the line is checked to see if the interrupt service routine has
; signalled. If so the various conditions are checked and the appropriate
; actions taken. As a result of an ISR signal a line may become "idle" and,
; therefore, eligible for a new operation.
;
; If the ISR has not signalled, a there is code to see if the line is idle.
; If the line is not idle we go on to the next line. If the line is idle
; we look for an operation waiting in the line's subq. The only operations
; that are placed onto the line subq are reads and writes. Other commands
; are processed directly by the main handler process.
;-
;
;+ Check for Data Set Interrupt
;-
assum$  I.FLAG EQ 0
BIT     #FSDEAT, R3           ; Check for data set attention
BEQ     40$                 ; Branch if not set
        ;+white branches if no modem control enabled
        ;-
MOV     R3,RI              ; Compute > head of report status change queue
ADD     #I.SQUE,RI           ;-
30$:   MOV     (R1),RI           ; Link toward > to next packet
        BEQ     40$                 ; If no more packets
        MOV     RI,RS              ; Compute > signal ISR
        ADD     #DP.SGL,RS           ;-
        BIC     #EMSBGM,DP.RJNVR  ; Must be a queue semaphore
        MOV     I.HBUF(R3),RM       ; Return receiver buffer
        MOV     I.RCSR(R3),D2.RPC(R1) ; Return receiver physical state
        MOV     XTSR(R3),DP.XPT(R1) ; Return transmitter physical state
        assum$ DP.TYP EQ <DP.TLS>? ; Alicreny?
        assum$ I.TYPE EQ <I.LUAF>? ; Alicreny?
        MOV     I.CLAS(R3),DP.CLS(R1) ; Return device class and type
        assum$ I.FLAG EQ 0
        MOV     (R3),DP.RGS(R1) ; Return receiver software state
        assum$ I.FLAG EQ 0
        MOV     @I.XFLG(R3),DP.XSTR(R1) ; Return transmitter software state
        assum$ ISSNR EQ 0
        CLR     DP.STS(R1)           ; Send back success code
        CALL    REPLY                ; Notify user
        BR      30$                 ; Loop for all packets

;+ Data Attention Request Processing:
; Data attention requests may mean that the transaction is complete, an error
; has been detected, a user signal counter has reached zero, etc. If the
; transaction was completed the line will be set to idle so that another
; operation can be started if any is waiting.
;
; A data attention request can occur along with a line set attention request.
;-
40$:   assum$ I.FLAG EQ 0
        BIT     #FSATTN, R3           ; Check for attention request
        BEQ     110$                 ; Branch if not attention

```

## SAMPLE DEVICE HANDLERS

```

50$:    MOV     I.LQLM(R3),R4      ; R4 -> Request packet
        BEQ     100$                ; Branch if no request packet
        MOV     R4,R0                ; Copy address of packet
        ADD     #DP.SEM,R0          ; Point to reply SDB in packet
        TST     (R0)                ; Check for semaphore SDB pointer
        BEQ     80$                ; Branch if no SDB
        BIT     #FM$BSM,DP.FUN(R4)  ; Test if binary semaphore
        BNE     70$                ; Branch if so, simply signal it

        MOV     DP.LEN(R4),DP.ALN(R4); Set Actual length transferred
        SUB     I.CNT(R3),DP.ALN(R4); original length - remaining length
assum$  IS$NOR EQ 0
        CLR     DP.STS(R4)          ; Assume normal completion
        TST     I.ERR(R3)           ; Check for error
assum$  ESERR EQ 100000
        BPL     60$                ; Branch if there are no errors
        MOVB   #ISSCTL,DP.STS(R4)  ; Set code to "Controller error"
        CMP     #ESERR!ESDPTY,I.ERR(R3); See if the only error is parity
        BNE     60$                ; There is some other error
        MOVB   #IS$PAR,DP.STS(R4)  ; The only error is parity. We have a
                                ; special error code for parity error

        assum$ I.FLAG EQ 0
        MOV     @R3,DP.ERR(R4)      ; Copy line status bits
        BIC     #^C<F$DONE!F$DSAT!F$IDLE>,DP.ERR(R4); Clear irrelevant status bits
        BIS     I.ERR(R3),DP.ERR(R4); OR in hardware/software error bits
sglq$  R0,R4                ; Send packet to user's Q semaphore
        BCC     90$                ; Br if no error and set line idle
        BR      80$                ; On error, simply free the packet

70$:    sgn1$  R0                 ; Signal the user's binary semaphore
                                ; Ignore any errors

80$:    dapk$  R4                 ; Now return packet to kernel pool

90$:    CLR     I.LQLM(R3)          ; Indicate no current packet
assum$  I.FLAG EQ 0
100$:   BIC     #F$ATTN!F$DONE,(R3); Clear attention and done bits
assum$  I.FLAG EQ 0
        BIS     #F$IDLE,(R3)        ; Flag line as avail for new work
        .BR     110$                ;+
;+      ; Enter here if line was not requesting attention. If it is idle check for a
;+      ; request in its sub-Q and if there is one start it going.
;+      ;+
110$:   assum$ I.FLAG EQ 0
        BIT     #F$IDLE,(R3)        ; Check "Idle" bit
        BEQ     130$                ; Branch if line is still busy

120$:   MOV     I.RQUE(R3),R4      ; Test if any packets waiting
        BEQ     130$                ; Branch if none
        MOV     (R4),I.RQUE(R3)      ; Else unlink it
        MOV     R4,I.LQLM(R3)        ; Save pointer to packet in impure
        CLR     I.ERR(R3)           ; Clear old error bits

        MOV     DP.FUN(R4),R1        ; Pick up command code passed
        BIC     #^CFNCOD,R1          ; Clear modifier bits
        ASL     R1                  ; Make word index
        JMP     @REQTBL(R1)          ; CASE (DP.FUN) OF

pdat$ 

```

## SAMPLE DEVICE HANDLERS

```

REQTBL: .WORD    180$      ; Read physical
        .WORD    180$      ; Read logical
        .WORD    210$      ; Read virtual (illegal)
        .WORD    180$      ; Write physical
        .WORD    180$      ; Write logical
        .WORD    210$      ; Write virtual (illegal)
        .WORD    210$      ; Set
        .WORD    210$      ; Get
        .WORD    160$      ; Connect receive ring buffer
        .WORD    140$      ; Connect transmit ring buffer
        .WORD    210$      ; Disconnect receive ring buffer
        .WORD    210$      ; Disconnect transmit ring buffer
        .WORD    210$      ; Get status after waiting

pure$  

130$:  JMP     200$  

;+  

; Handle connect transmit ring buffer  

; Create the helper subprocess, its semaphore, and pass it parameters in  

; its stack  

;-  

140$:  MOV     R3,R5      ; Copy impure pointer
        ADD     #I.HSDB,R5      ; Point at helper semaphore sdb slot
        MOV     R5,I.HLSM(R3)   ; Set pointer to it in impure area
        CLR     (R5)          ; Clear ID field for restart
        CLR     I.HSNM(R3)    ; Set null name
        crst$  sdb=R5,styp= ST.BSM,satr= SA$OPR,value= 0
        BCC     150$          ; Br if no error
        CALL    $DEGCS        ; Report exception (Does not return)

150$:  ADD     #I.HPDB-I.HSDB,R5 ; Point at helper process pdb slot
        CLR     (R5)          ; Clear ID field for restart
        CLR     I.HPNM(R3)    ; Set null name
        MOV     R3,I.XBP(R3)   ; Compute -> output burst mode buffer
        ADD     #I.XBUF,I.XBP(R3); "
        MOV     R3,R1          ; Set up stack pointer
        ADD     #I.GRD2,R1      ; -> Guard word
        MOV     R3,R2          ; Compute stack guard word
        ADD     #I.GRD1,R2      ; "
        push$  <R4>          ; Save R4
        MOV     R1,R4          ; Upper bound of stack
        MOV     R3,-(R1)        ; Load stack with impure's location

        'rpos$ p1b=R5,pri= XLSPPR-1,cxos= CXSTD,trp= 1,her= DLHTRM, xl= 0,s*1=R1,s*1=R2,sth=21,start= DLHLP,in= '
        pop$   <R4>          ; Restore R4
        BCC     170$          ; Br if no error and finish up by sharing
                                ; Some code with connect receive ring buffer
        CALL    $DEGCP        ; Report exception (Does not return)

;+  

; Connect receive ring buffer  

; Store ring buf SDB in impure and flag line as busy with ring buf request  

; The SDB is passed in the DP.SEM field of the packet  

;  

; Initialize burst mode buffer control blocks  

;-  

160$:  MOV     #1,I.FRKS(R3) ; Enable fork
        MOV     R3,R1          ; Compute -> buffer control block
        ADD     #I.IBFL,R1      ; "
        MOV     R1,I.IBCB(R3)   ; Give ISR ownership of first buffer
        MOV     R1,R2          ; Compute -> alternate buffer control block
        ADD     #BE.SIZ,R2      ; "
        push$  <R2,R1>        ; Save for later
        CALL    BCBINI        ; Initialize buffer control block

```

## SAMPLE DEVICE HANDLERS

```

pop$    <R2,R1>          ; Swap current and alternate pointers
CALL    BCBINI           ; Initialize alternate buffer control block
MOV    #1,I.CNT(R3)      ; Set count in GELM/PELM parm block to 1 char
MOV    R3,I.DAT(R3)      ; Initialize string pointer to buffer word
ADD    #I.BUF,I.DAT(R3)  ;           "

; The following code is common to the connect to transmit ring buffer

170$:  MOV    DP.SEM+0(R4),I.RDB+0(R3) ; Copy ring buffer sdb into impure
       MOV    DP.SEM+2(R4),I.RDB+2(R3) ;
       MOV    DP.SEM+4(R4),I.RDB+4(R3) ;
       MOV    DP.FUN(R4),R0           ; Get function modifier bits
       BIC    #^CF$XCHK,R0          ; Isolate only the X-ON checking bit
assum$ I.FLAG EQ 0
       BIC    #FSXCHK,(R3)
assum$ I.FLAG EQ 0
       BIS    R0,(R3)              ; Store the modifier bits
       MOV    R3,I.RBF(R3)         ; Setup GELM/PELM arg block in impure for ISR
       ADD    #I.RDB,I.RBF(R3) ;
       CLR    I.LQLM(R3)          ; Clear any possible old packet
       dapk$ R4                  ; Now deallocate user's packet
assum$ I.FLAG EQ 0
       BIS    #FSRING,(R3)        ; Flag ring mode now in progress
       BR    190$                 ; Ring is set, go initiate I/O in common code

;+
; Common code to initiate block mode reads and writes
;
; In the mapped case, we bias the address to the beginning of PAR 1
; and adjust the PAR value to avoid doing a carry calculation at interrupt
; level
;-
180$:  MOV    DP.FUN(R4),R0           ; Get function modifier bits
       BIC    #^CF$XCHK,R0          ; Isolate only the X-ON checking bits
assum$ I.FLAG EQ 0
       BIC    #FSXCHK,(R3)
assum$ I.FLAG EQ 0
       BIS    R0,(R3)              ; Store the bits
       DRMAP$ DP.BUF(R4),DP.PAR(R4),I.ADR(R3),I.PAR(R3)
       MOV    DP.LEN(R4),I.CNT(R3) ; Byte count
       BNE    190$                 ; Br if non-zero length request
       JMP    50$                  ; Zero length request, signal user now

;+
; Initiate input/output by enabling interrupts on the respective line
;
; Read hardware "BUF". If it is an output line nothing happens. If it is
; an input line any noise character is cleared. Note that "type ahead"
; cannot be handled until some process has provided a place for the
; characters to go -- most likely a ring buffer.
;-
assum$ I.FLAG EQ 0
190$:  BIC    #F$IDLE,(R3)          ; Line is in use now, clear idle bit
       MOV    I.HBUF(R3),R0          ; Get address of hardware "BUF" register
       MOV    (R0),R1                ; Read just to clear input character
assum$ rCSINT EQ XCSINT
assum$ CSR EQ <RBUF - 2>
assum$ XCSR EQ <XBUF - 2>
       BIS    #RCSINT,-(R0)         ; Set interrupt enable
                                      ; interrupt will propagate transfer

;+
; Done with this line. Reset R3 to point to next line, if any, and go back
; to beginning of loop to count down and test for being done with all lines.
;-

```

## SAMPLE DEVICE HANDLERS

```

200$: MOV I.LINK(R3),R3           ; Get pointer to next line table
      JMP 20$                   ; Go check next line in list

;+
; Catch bad function codes; should not occur
;-

210$: HALTS

        .dsabl LSB

;+
; End of internal control process.
;-

DLCTRM: dlpc$                  ; Control process termination entry

;+
; BCBINI
;
;     Buffer control block initialization
;
; Input:
;
;     R1          -> Current BCB
;     R2          -> Alternate BCB
;
; Output:
;
;     R0,R1,R2      Modified
;-

BCBINI:
        MOV R1,R0             ; Compute -> buffer
        ADD #B.BUFF,R0         ;
        assum$ B.BUFP EQ 0      ;
        MOV R0,(R1)+            ; Initialize buffer pointer
        assum$ B.CCNT EQ B.BUFP+2   ;
        CLR (R1)+               ; Initialize character counter
        assum$ B.ABP EQ B.CCNT+2   ;
        MOV R2,(R1)              ; Initialize alternate buffer pointer
        RETURN
        .sbttl Ring buffer output helper process

;+
; Transmit ring buffer helper process
;
; This process allows the output ISR to look as though it may block
; on its ring buffer when a character is not immediately available
;
; INPUT:
;     SP -> End of line's impure area
;     (SP) = Start of impure area
;-

DLHLP: MOV (SP)+,R3            ; Pick up pointer to impure passed in stack
      MOV I.HBUF(R3),R4        ; Get pointer to XBUF
      MOV R3,R2                ; Now create parameter block pointer for GELM
      ADD #I.RBF,R2

10$:  MOV R3,R0                ; Create parameter block pointer for WAIT
      ADD #I.HLSM,R0
      wait$                   ; Wait for ISR to "block" on GELM
      BCC 20$                 ; Br if no error
      CALL SDESID              ; Report exception (Does not return)

```

## SAMPLE DEVICE HANDLERS

```

20$:    MOV     #1,I.CNT(R3)      ; Initialize gelm$ parameter block for
        MOV     I.XBP(R3),I.DAT(R3) ; a single character beginning at I.XBUF
gelm$    R2                  ; Evoke a synchronous GELM
BCC     30$                ; Br if no error
CALL    SDESID              ; Report exception (Does not return)

30$:    MOVB   @I.XBP(R3),(R4) ; Load XBUF with character to start the ball rolling
        BIS    #XC$INT,CSR(R4) ; Enable interrupts
        BR     10$                ; Repeat indefinitely

DLHTRM: dlist$  I.HLSM(R3)      ; Delete the helper semaphore
                                         ; Ignore any errors
        dlpc$ .sbttl Input Interrupt Service Routine
        .dsabl ama               ; ISRs MUST be PIC

;+
; Input interrupt service routine
;
; On entry to the ISR:
;
; R3      -> ISR impure data area
; R4      -> RBUF (Receiver data buffer)
;-

$IIISLD:: assum$ I.FLAG EQ 0
            BIT    #FSDSAC,(R3)      ; Test if modem control enabled
            BEQ   10$                ; Branch if not
            TST    CSR(R4)          ; Now test if modem changed state
            BPL   10$                ; Branch if not
assum$ I.FLAG EQ 0
            BIS    #F$DSAT,(R3)      ; Set dataset attention to notify user
            BR     20$                ; Save RCSR contents and exit

10$:    TSTB   CSR(R4)          ; Now check if character came in
            BPL   50$                ; Branch if no character, just dismiss interrupt
            MOV    (R4),I.CHAR(R3) ; Else capture character, save in impure block
            BPL   30$                ; Branch if no errors
assum$ I.FLAG EQ 0
            BIT    #FSRING,(R3)      ; Connected to a ring buffer?
            BNE   30$                ; Br if so and ignore error. No good way
                                         ; to report it.
            BIS    #FSCTRL,(R3)      ; Set controller error indicator
            MOV    I.CHAR(R3),I.ERR(R2) ; Copy error bits
            CLRB  I.ERR(R3)          ; Hardware errors are in the high byte only
20$:    MOV    RCSR(R4),I.RCSR(R3) ; Save contents of receiver CCR
            BR     SIGFRK             ; Notify user

assum$ I.FLAG EQ 0
30$:    BIT    #FSXCHR,(R3)      ; Test if checking for X-ON/X-OFF
            BEQ   60$                ; Branch if bypassing this test
            MOV    I.CHAR(R3),-(SP) ; Copy the character just received
            BIC    #C177,(SP)          ; Take only 7 bits to ignore parity
            SUB    #XONCHR,(SP)        ; Subtract value of X-ON
                                         ; (FP) is 1 if X-ON hit, 0 if X-OFF hit, <0
assum$ XOFCHR EQ XONCHR+?
            ROR    (SP)                ; Shift X-ON/X-OFF bit to carry
            ROR    (SP)+               ; Test and discard stack
            BNE   60$                ; Branch if not X-ON or X-OFF
            BCC   40$                ; Branch if X-ON
            BIS    #FSINHO,@I.XELG(R3) ; X-OFF: set interrupt flag in interrupt
            RETURN                      ; and dismiss interrupt

```

## SAMPLE DEVICE HANDLERS

```

40$: BIC    #FSINHO, #I.XFLG(R3) ; Clear inhibit output bit in impure
      BIS    #XCSINT, XCSR(R4) ; and set interrupt enable, the
                                ; interrupt will propagate any further output
50$: RETURN

60$:
assum$ I.FLAG EQ 0
      BIT    #FSIDLE, (R3)   ; Test if idle line
      BNE    EXIT          ; Branch if so, just dismiss interrupt
assum$ I.FLAG EQ 0
      BIT    #FSRING, (R3)   ; Test if in ring buffer mode
      BEQ    120$           ; Branch if not, handle block mode input
      MOV    I.IBCB(R3), R4  ; -> Burst mode buffer control block
      CMP    B.CCNT(R4), #BL.17E ; Burst mode ring buffer full?
      BEQ    108$           ; Br if so
assum$ B.BUFF EQ P
      MOVB  I.CHAR(R3), @ (R4) ; Copy character to burst mode buffer
assum$ B.CCNT EQ B.BUFF+2
      INC    (R4)           ; Increment character counter
assum$ B.BUFF EQ B.CCNT-2
      INC    -(R4)          ; Increment buffer pointer
      ASRB  I.FRKSI(R3)    ; Fork in progress?
      BCC    EXIT          ; Br if so
      MOV    B.ABP(R4), I.IBCB(R3) ; Switch to alternate buffer
      forks #KLSFPR          ; Spawn fork process to dump buffer
      BCS    108$           ; Br if overrun, a bug if this happens
      MOV    R4, R0           ; Compute -> buffer
      ADD    #B.BUFF, R0
      MOV    R0, I.DAT(R3)
      MOV    R0, B.BUFF(R4)   ; Reset buffer pointer
      MOV    B.CCNT(R4), I.CNT(R3) ; Copy number of characters in buffer
      CLR    B.CCNT(R4)     ; Reset character counter
      MOV    #I.RBF, R0       ; Point at RBF parameter line in impure
      ADD    R3, R0           ; Absolute
      IOT    SSpelc          ; Execute the primitive via special entity which
                                ; does no CPU checking and dump the burst mode
                                ; buffer into the ring buffer
      TST    R0               ; Test if had room in ring
      BEQ    80$              ; Branch if so
      BIS    #FSATTN, R3     ; Else set attention to log the overrun
      BIS    #ESERR|ESP.VR, I.FRR(R3) ; and set error bits
      SPL    KLSHPR          ; Disable further device interrupts
                                ; with buffers
      MOV    I.IBCB(R3), R0  ; -> current IBC buffer control block
      MOV    R4, I.IBCB(R3)  ; -> alternate buffer control block
      MOV    R2, R4           ; -> Fork process buffer control block
      TST    B.CCNT(R4)     ; Any data in the buffer?
      BEQ    90$              ; Br if not
      SPL    R0
      BR    70$              ; Enable interrupts

90$:
      INCR  I.FRKI(R3)    ; * Enable task
      SPL    R0
      RETURN             ; * Enable interrupt pins
                                ; Exit fork process

100$:
assum$ I.FLAG EQ 0
      BIS    #FSATTN, R3     ; Else set attention to log the overrun
      BIS    #ESERR|ECONVR, I.FRR(R3) ; and set error bits
      RETURN             ; Exit the fork process

110$:
      TST    I.INT(R3)       ; Test if count exhausted
      BEQ    EXIT            ; Branch if so, ignore the char
      MVBYTS I.BUF(R3), @I.ADR(R3), I.FAR(R3) ; store character in user buffer
      IADRGS I.ADR(R3), I.PAR(R3)           ; Increment user buffer pointer
      DFC    I.CNT(R3)       ; Increment cycle count
      BNE    EXIT            ; Branch if count not exhausted

```

## SAMPLE DEVICE HANDLERS

```

SIGFRK: BIS      #FSDNIEPCATTN,0R1      ; Set flags
        XRS      #XISPR
        BCS      XMTDVR
; set flags
; call spawn task, if none
; do it FOREVER

SIGSEM: MOV      #I, ITSM, R2
SIGHELP: ADD     R3, R2
        Sqn19
        BCC      EXIT
        CALL    SDESID
; Set argument block pointer and
; Input base address + offset
; Start an error
; Report exception to user
; If no error, then continue

XMTDVR: BIS      #FSATTN,(R3)      ; Set attention flag to system
        BIS      #FESERRIESXFOV,I,ERR,R3  ; Transmitter task overrun. Hardware
        EXIT
; Common exit, if no interrupt
; Output interrupt service routine

; Output interrupt service routine
; In normal block mode, characters are removed from the user's buffer. The
; control process is signalled when the buffer is emptied to signal the
; user of the event.
; If a ring buffer is connected to this line, characters are pulled from
; it instead using a SELM primitive.
; On entry to the ISR:
; R3      -> ISR data area
; R4      -> XBUF (transmitter data buffer)
;-

SOISLD:
assums I,FLAG,EQ,0
BIT    #FSINH?,R3
BNE   20$           ; Test if X-BUF was hit
; If so, just dismiss the interrupt
; X-ON will generate interrupts later

assums I,FLAG,EQ,1
BIT    #FSRING,R3
BNE   40$           ; Test if ring buffer mode
; Branch if so, handle separately

assums I,FLAG,SL,1
BIT    #FSIDLE,(R3)
BNE   20$           ; Test if idle line
; Branch if so, ignore the interrupt
; Test if count is exhausted now
; Branch if so, signal control process
; which will then signal user

assums RBUF,EQ,0
MVBYTS #I,ADR(R3),#R4,I,PAR(R3) ; Send a character
IBADRS #I,ADR(R3),I,PAR(R3)       ; Increment user buffer pointer

DEC    I,CNT(R3)          ; Decrement byte count

10$:  RETURN           ; Dismiss the interrupt

20$:  BIC    #XCSINT,ISR(R4) ; Clear interrupt enable
        RETURN           ; and dismiss interrupt

30$:  BIC    #XCSINT,CSR(R4) ; Clear interrupt enable
        BR    SIGFRK      ; Dismiss interrupt and signal control
                           ; process

; Handle ring buffer mode interrupts
;
; We attempt to output another character. If no char is available, the
; associated "helper" process is stimulated to load the buffer for us.

```

## SAMPLE DEVICE HANDLERS

```
; There the XBUF is loaded with the next string of characters. The
; resulting interrupt will return here where (hopefully) further
; characters will be ready for output.
;-

40$: DEC    I.CNT(R3)          ; Decrement character count
      BLE    60$              ; Br if no more characters to output
50$: MOVB   @I.DAT(R3), (R4)  ; Output the next character
      INC    I.DAT(R3)         ; Increment the buffer pointer
      RETURN                         ; Dismiss the interrupt

60$: fork$  #XL$FPR           ; Spawn fork process
      BCS    XMTOVR           ; Br if FORK overrun
      MOV    I.XBP(R3), I.DAT(R3) ; -> First empty space in the buffer
      MOV    #IO.BSZ, I.CNT(R3)  ; Attempt to fill the buffer
      MOV    R3, R0              ; Compute -> gelc$ parameter block
      ADD    #I.RBF, R0          ;
      IOT    $SGELC             ;
      SUB    R0, I.CNT(R3)       ; Reduce the character count by the
                                ; number we didn't get
      BNE    50$              ; Br if we got one or more

70$: BIC    #XC$INT, CSR(R4)  ; Clear interrupt enable
      MOV    #I.HLSM, R0          ; Set offset to helper semaphore SDB
      BR     SIGHLP             ; Signal it and dismiss the interrupt

.end
```

## SAMPLE DEVICE HANDLERS

### A.3 DRV11 (YA) HANDLER--SAMPLE PASCAL DEVICE-HANDLER SOURCE

{

COPYRIGHT (c) 1982 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.

}

```
[ system( MicroPower ), data_space( 2000 ), priority( 248 ),  
stack_size( 500 ), privileged ]
```

PROGRAM drv11;

[

Module: YADRV11.PAS ( DRV11 device driver for mapped system)  
Note: Requires (includes) IOREQ1.PAS, DRVDEF.PAS, GETPUT.PAS  
for compilation of this module; and, YAPFXM.PAS must be  
separately compiled and merged with YADRV11 to build a  
DRV11 driver process.

Author: JAS

Date: 10-August-1981

Modifications:

Definition of Function:

This driver process provides 16 bit block mode input and output for the DRV11 device. It will control one such device whose CSR and vector are defined by the InitCSR procedure supplied in the YAPFX.PAS prefix module and called from the driver initialization procedure.

The DRV11 is a single unit, bi-directional, 16-bit binary transfer device with separate read/write interrupt enables.  
A typical device configuration for the first module is as follows:

CSR	167770
OUTBUF	167772
INBUF	167774

Write Interrupt(REQ A) Vector = 300  
Read Interrupt (REQ B) Vector = 304

Input:

Request messages are sent to the DRV11 driver in the standard driver request format using the SEND directive. Only the function code, function code modifiers, transfer length, user buffer and reply semaphore fields are used. There are no device dependent function modifiers. The transfer count is specified in bytes although the device transfers 16 bits at a time; thus, for expected operation the byte count should be even. If not, it is rounded down to an even number of bytes --e.g., a 3 byte request results in the transfer of 1 word, a 7 byte request in 3 words, etc.

## SAMPLE DEVICE HANDLERS

Note that the include file IOREQI.PAS defines the IO\_REQUEST record for "internal" usage, i.e., by the driver process (GET/PUT packet formats). The IOREQX.PAS defines the appropriate "external" definitions of the request packet for users of the driver (SEND/RECEIVE formats).

### Output:

Upon completion of the user's request, or if an error was made in the input request, a queue packet is sent to the user's reply semaphore (if one was specified)--if a binary or counting semaphore was specified, it is signalled. The reply packet contains the number of bytes transferred (always 0 or the number specified in the request), the status code (always normal or illegal command), and the information in the original request packet. Data is returned directly to or removed from the user's buffer (address) specified in the request packet.

Note that the IOREQI.PAS and IOREQX.PAS include files define the IO\_REPLY record.

### Process organization:

The [INITIALIZE] procedure for the driver process creates the following semaphores:

SYAA Queue semaphore for all driver requests.  
SYAARQ Queue semaphore for reads.  
SYAAWQ Queue semaphore for writes.  
SYAARI Binary semaphore signalled by read ISR.  
SYAAWI Binary semaphore signalled by write ISR.

The static process starts at priority 248 and then creates two dynamic processes, as follows:

DRV11\_IO ( NAME := 'DRARDR', PRIORITY := 249 )  
DRV11\_IO ( NAME := 'DRAWRT', PRIORITY := 249 )

The static process (running at priority 248) is preempted when each of above dynamic processes is created (at priority 249); therefore, each of the above runs until it's about to wait on its appropriate request queue; and, at that point, reduces its priority to the value specified by the variable STDPRI0 (the standard priority for drivers of DRV11 category).

Note that the process code for DRV11\_IO is shared by the two processes DRARDR and DRAWRT but that each has its own data context.

The static process then lowers its priority to STDPRI0 and "becomes" the dispatcher process waiting for request packets on the SYAA queue semaphore. YAARDR will start and wait on its read request queue semaphore, SYAARQ, and, YAAWRT starts and waits on its write request queue semaphore, SYAAWQ. Note that [INITIALIZE] code for the driver process will have been executed even before the static process so that all queue and binary semaphores will have been created before any requests could have been generated.

When a request is received the Dispatcher validates the function. If an illegal function is detected, and a reply queue semaphore had been given, a reply packet is returned to the requestor with an illegal function (IS\$FUN) status. If a binary or counting semaphore had been specified for replies, that semaphore will be signalled. If a valid read or write request is received, the Dispatcher sends the request packet on to either the YAARDR or YAAWRT process as appropriate.

The YAARDR (reader) process, through cooperation with the interrupt handling of the kernel which signals the SYAARI binary semaphore each time a word of data is available in the device register, effects the transfer of data from the device data register into the user's buffer. When the transfer is complete (as specified by the byte count) the reader process sends an IO\_REPLY back to the requestor and then returns

## SAMPLE DEVICE HANDLERS

```
to wait on its input queue. The YAAWRT (writer) process acts in similar
fashion, using the SYAAWI semaphore, to transfer the specified number
of words from the user's buffer to the output device.
}

INCLUDE 'DRVDEF'

{ Status code returned in high-order 13 bits of STATUS field of reply rcd.}
CONST
  ISSNORMAL = 0;           {Normal, successful status}
  ISSFUNCTION_INV = 9;     {Illegal function/command status}

{ Severity level returned in low-order 3 bits of STATUS field.}
  IOSUCCESS = 0;
  IOSWARNING = 1;
  IOSINFO = 2;
  IOSERROR = 3;
  IOSFATAL = 5;           {Defines IO status/error codes,etc.}
INCLUDE 'IOREQI'
{+ Include section DRREQI.PAS defines the IO-request and IO-reply
packet formats for internal use by the DRVII driver, using
GET_PACKET and PUT_PACKET requests (full 40 byte packets
are mapped by the record definitions). NOTE: DRREQX.PAS should
be used by "external" users (non-privileged) who will issue
SEND/RECEIVE directives.
Author: JAS (Jon A. Stewart)
Date: 28-Sept-1981
-} x

CONST
  max_funct_code = 63;

{ Define assigned function codes }

  read_physical = 0;
  read_logical = 1;
  rsvd2 = 2;
  write_physical = 3;
  write_logical = 4;
  rsvd5 = 5;
  set_characteristics = 6;
  get_characteristics = 7;

TYPE
  funct_code = 0..max_funct_code;

  fun_ind_bits = (simple_reply,
                  data_chk,
                  noretry);
  fun_dep_bits = (bit0, bit1, bit2, bit3, bit4, bit5, bit6);

  block = ARRAY[0..255] OF unsigned;
  block_ptr = ^ block;

  device_address =
PACKED RECORD
  CASE funct_code OF
    read_physical,
    write_physical : (sector, track : byte;
                      cylinder : unsigned);
    read_logical,
    write_logical : (block_num : longint);
  END;
```

## SAMPLE DEVICE HANDLERS

```
io_req =  
PACKED RECORD  
    link,                      (+      )  
    aux_link,                 { header portion OF packet}  
    priority : integer;       (-      )  
    oper : [pos(48),bit(6)] funct_code; { operation }  
    dep_mod : [pos(54),bit(7)] PACKED SET OF fun_dep_bits;  
    ind_mod : [pos(51),bit(3)] PACKED SET OF fun_ind_bits;  
    unit_num,                  { unit number }  
    filler_1 : byte;          { reserved }  
    sequence : integer;        { user specified sequence number}  
    CASE funct_code OF  
        read_physical,  
        read_logical,  
        write_physical,  
        write_logical :  
            (pid,                     { requestor's PDB }  
             reply_sem : structure_id; { Reply semaphore SDB }  
             dev_addr : device_address; { Device address }  
             buffer_ptr : physical_address; { Buffer address-- wds }  
             buf_len : unsigned); { 2nd is PAR value }  
             { Length of buffer }  
    set_characteristics : ();  
  
    get_characteristics :  
        (filler_2 : ARRAY [1..3] OF integer;  
         class,  
         dev_type : byte;  
         blocks : longint;  
         sectors,  
         tracks : byte;  
         cylinders : integer)  
    END;  
  
io_reply =  
PACKED RECORD  
    link,                      (+      )  
    aux_link,                 { header portion OF packet}  
    priority : integer;       (-      )  
    oper : [pos(48),bit(6)] funct_code; { operation }  
    dep_mod : [pos(54),bit(7)] PACKED SET OF fun_dep_bits;  
    ind_mod : [pos(51),bit(3)] PACKED SET OF fun_ind_bits;  
    unit_num,                  { unit number }  
    filler_1 : byte;          { reserved }  
    sequence : integer;        { user specified sequence number }  
    status,  
    act_length,  
    err_code : integer;  
    reserved : ARRAY [1..3] OF integer;  
    dev_addr : device_address;  
    class,typ,subtype,filler_2:byte;  
    fun_dep_dat : ARRAY [1..3] OF integer;  
    END;  
  
io_req_ptr = ^ io_req;  
  
io_reply_ptr = ^ io_reply; {Defines IO REQUEST, IO REPLY records  
{ for "internal" use by the driver with  
{ GET/PUT packet requests.}  
  
#INCLUDE 'GETPUT'  
{ Module: GETPUT.PAS }
```

## SAMPLE DEVICE HANDLERS

{  
\* These procedures are provided as a service for device drivers written in MicroPower/Pascal. The map\_parl\_addr procedure is provided to convert the mapped addresses to a form usable by the GET\_x and PUT\_x procedures. The GET\_x procedures are for input. The specified byte or word is copied to the location designated by the PAR 1 address. The PUT\_x procedures are for output. A byte or word is copied from a location indicated by the PAR 1 address to the specified byte or word.  
\* }

```
TYPE  
  parl_addr = PACKED RECORD  
    parl_addr : UNSIGNED;  
    par_value : UNSIGNED;  
  END;
```

{  
\* This procedure converts a standard "PHYSICAL ADDRESS", i.e., a virtual address and a PAR value to a PAR 1 address and PAR value that can be used by the GET\_x and PUT\_x procedures. The "PHYSICAL ADDRESS" is that returned in the "RET\_INFO" argument of a RECEIVE procedure call.  
\* }

```
[EXTERNAL (SMPARI)] PROCEDURE map_parl_addr (x : PHYSICAL_ADDRESS;  
                                              VAR y : parl_addr);  
EXTERNAL;
```

{  
\* Parameters:  
\* }

x                   A physical address consisting of a virtual address and its PAR value.

y                   The physical address remapped to a PAR 1 address and its PAR value.

\* }

{  
\* Because of their similarities, the following procedures are discussed as a group. The procedure name indicates whether a byte or a word is to be moved. The procedure name prefix indicates the direction of the move. A GET\_x procedure is for input from a device to a buffer. A PUT\_x procedure is for output to a device from a buffer. In all procedures, the PAR1\_ADDR is updated after the move so that it points to the next element in the buffer.

Assumptions:

1. The process is executing in USER mode and appended to SPNEN1 or SUPERVISOR mode.
2. The process is privileged and mapped to the file page.

Side effects:

The user PAR 1 and PDR 1 registers are modified and not restored.

\* }

```
[EXTERNAL (PUTBYT)] PROCEDURE put_byt (par1_addr : WORD;  
                                         VAR par1_addr);  
EXTERNAL;
```

```
[EXTERNAL (PUTWORD)] PROCEDURE put_word (par1_addr : WORD;  
                                         VAR par1_addr);  
EXTERNAL;
```

## SAMPLE DEVICE HANDLERS

```
      EXTERNAL;
      [EXTERNAL (SGETBT)] PROCEDURE get_byte( x : BYTE;
                                              VAR y : parl_addr );
      EXTERNAL;

      [EXTERNAL (SGETWD)] PROCEDURE get_word( x : INTEGER;
                                              VAR y : parl_addr );
      EXTERNAL;

{*
Parameters:
  x           Byte or word, as specified in the declaration. In the
              case of a GET_x, it is the source. For PUT_x
              procedures, it is the destination.
  parl_addr   This is a PAR 1 address and its PAR value. After the
              move, the PAR 1 address and its PAR value updated to
              point to the next element to be transferred.

Side effects:
  PAR 1       The User Instruction PAR 1 is modified
  PDR 1       The User Instruction PDR 1 is modified
*}
      (*)          (Defines procedures Map-parl-addr,
                  Put_word and Get_word)

CONST
  max_buf_size = 256;           {Formal--limit ignored by driver}

TYPE
  io_buffer = ARRAY[ 1.. max_buf_size ] OF unsigned;
  io_buf_ptr = ^ io_buffer;

  d.v11_csr = PACKED RECORD
    forcea: [pos(0),bit] boolean; {simulates interrupt}
    forceb: [pos(1),bit] boolean; {simulates interrupt}
    intenb: [pos(5),bit] boolean;
    intena: [pos(6),bit] boolean;
    reqa: [pos(7),bit] boolean;
    reqb: [pos(15),bit] boolean;
    outbuf: [pos(16),word] unsigned;
    inbuf: [pos(32),word,volatile] unsigned;
  END;

VAR
  inbuf_ptr: ^ integer;
  outbuf_ptr: ^ integer;
  req_ptr, io_req_ptr; {Type defined in IOREQI.PAS}
  stop: boolean;        {Used to shut down the driver}
  version:[ global(SYAVER) ]PACKED ARRAY[1..6] OF char;
{NOTE: This is the unresolved global
 from YAPFX which "pulls in" the
 driver from the appropriate library}

{Following defined in the YAPFX module and
 initialized by initCSR call}

  csr :[ EXTERNAL ] ^ d.v11_csr;
  drprio:[ EXTERNAL,readonly ] integer;
  stdprio:[ EXTERNAL,readonly ] integer;
  vector_reqa:[ EXTERNAL(reqa) ] unsigned;
  vector_reqb:[ EXTERNAL(reqb) ] unsigned;
  maintenance:[ EXTERNAL(maint) ] boolean;
  mapped:[ EXTERNAL ] boolean;
```

## SAMPLE DEVICE HANDLERS

```
[initialize]
PROCEDURE initdrv11;
BEGIN
    IF NOT create_queue_semaphore( name:='$YAA' )
    OR
    NOT create_queue_semaphore( name := 'SYAARQ' ) {Read request}
    OR
    NOT create_queue_semaphore( name := '$YAAWQ' ) {Write request}
    OR
    NOT create_binary_semaphore( name := '$YAAIA' ) {Interrupt A}
    OR
    NOT create_binary_semaphore( name := '$YAAIB' ) {Interrupt B}
    THEN stop := true
    ELSE stop := false;
END;
{Procedure InitDRV11}

PROCEDURE reply_to_requestor( request_ptr: io_req_ptr;
                             return_code: integer;
                             severity: integer );
VAR
    reply_desc: semaphore_desc;
    reply_ptr: io_reply_ptr;
BEGIN
    reply_desc := request_ptr^.reply_sem::semaphore_desc;
    reply_ptr := request_ptr::io_reply_ptr;
        {Use the same packet for reply.}
    reply_ptr^.err_code := return_code;
    reply_ptr^.status := return_code * 8 + severity;
    IF simple_reply IN request_ptr^.ind_mod
    THEN signal( desc := reply_desc )
    ELSE put_packet( desc := reply_desc,
                     packet_ptr := reply_ptr::queue_ptr );
END;
{Reply_to_requestor}

[ stack_size( 200 ), context( MMU ) ]
PROCESS drv11_io( io_mode:char );

{This process code is used with by both the YAARDR (reader) process
and the YAAWRT (writer) process. On creation of each of the processes
local variables (separate copies for each process) are initialized
for both the name of the interrupt binary semaphore, INT_NAME, and
the name of the request queue semaphore, QUE_NAME. The IO_MODE parameter
('R' for reader, 'W' for writer) determines whether a "reader" or
a "writer" process is being created.

The process waits on read or write requests from the Dispatcher which are
sent to QUE_NAME. It will process each request in entirety before
going on to the next request (queued in the meantime, if any).
When the specified number of words have been transferred, a reply
packet will be sent back to the requestor (if reply-semaphore given).
}

VAR
    request: io_req_ptr;
    index, byte_count, word_count: integer;
    buf_ptr: ^ io_buffer; {Unmapped uses VA from physical_addr of request}
    buf_physl_addr: physl_addr;
    {Physical address converted to physl_addr
    type, by call to Map_physl_addr for mapped version}
```

## SAMPLE DEVICE HANDLERS

```
que_name: PACKED ARRAY[1..6] OF char;
int_name: PACKED ARRAY[1..6] OF char;
int_desc: semaphore_desc;
reading: boolean;
int_vector: unsigned;

BEGIN      {Following process specialization is only done once}
{ after the reader/writer process is created and started.}

    que_name := '$YAA Q';
    que_name[ 5 ] := io_mode;
    IF io_mode = 'R'
        THEN
            BEGIN
                reading := true;
                int_name := '$YAAIB' ; {DRV11 Interrupt REQ B}
                int_vector := vector_reqb;
            END
        ELSE
            BEGIN
                reading := false;
                int_name := '$YAAIA' ; {DRV11 Interrupt REQ A}
                int_vector := vector_reqa;
            END;
    init_structure_desc( desc := int_desc,
                         name := int_name );
{NOTE: Already created by name.
This sets up descriptor for
more efficient access.}

    connect_semaphore( vector := int_vector,
                       ps   := drprio,
                       desc := int_desc
                     );

    change_priority( stdprio );      {Lower to standard priority for DRV11}

    WHILE NOT stop
        DO                      {Head of request processing loop}
        BEGIN
            get_packet( name := que_name,
                        packet_ptr := request::queue_ptr);

            byte_count := request^.buf_len;
            IF mapped
                THEN map_parl_addr( request^.buffer_ptr, buf_parl_addr )
                ELSE buf_ptr := request^.buffer_ptr::io_buf_ptr; {only the VA used}

            word_count := byte_count DIV 2;           {Note: integer division}

            FOR index := 1 TO word_count
                DO
                    BEGIN
                        IF reading
                            THEN
                                BEGIN
                                    wait( desc := int_desc);
                                    {Wait for interrupt, 16 bits available.}
                                    IF mapped
                                        THEN get_word( inbuf_ptr^, buf_parl_addr )
                                        ELSE buf_ptr^[ index ] := csr^.Tnbuf;

                                    {Move word from the device register to the user buffer.}
                                    {NOTE: With loopback can now let the "writer" go ahead.}
                                END;
                            END;
                        END;
                    END;
                END;
            END;
        END;
    END;
```

## SAMPLE DEVICE HANDLERS

```
IF maintenance
THEN
BEGIN
  csr^.forcea := false;
  csr^.forcea := true;
  {Must toggle to force interrupt--set REQA;
   for debug only with loopback cable.}
END;
END
ELSE    {Writing}
BEGIN
  IF mapped
  THEN  put_word( outbuf_ptr^, buf_parl_addr )
  ELSE  csr^.outbuf := buf_ptr^ [ index ];

  {Move word from the user buffer to the device register.
  NOTE: With loopback this means the INBUF is new, too,
  Therefore, force an INPUT interrupt.}

  IF maintenance
  THEN
  BEGIN
    csr^.forceb := false;
    csr^.forceb := true;
  END;

  wait( desc := int_desc );
  {Here when OUTPUT done.}
END;
{ Completed the request; "signal" a reply.}

reply_to_requestor( request, is$normal, io$success );
{NORMal, successful return.}
END;
{Request processing loop}

END;
{Process DRV11_IO}

PROCEDURE initcsr; EXTERNAL;      {Defined in YAPFX module}

BEGIN  {Static process for DRV11 driver--note: PRIORITY = 248}

{Create the dynamic processes}

initcsr;                                { Sets up CSR and VECTOR addresses
                                             and defines VERSION of the
                                             YAPFX module. }
inbuf_ptr::unsigned := csr::unsigned + 4;  {Address computations}
outbuf_ptr::unsigned := csr::unsigned + 2;

IF ( version <> 'Y01.01')    { VERSION of the YDRV module
                               should match that of YAPFX }
OR
stop          { STOP = true if failed to get
               structures }

THEN stop := true
ELSE { OK to proceed }
```

## SAMPLE DEVICE HANDLERS

```
BEGIN

    drvll_io( name := 'YAARDR', priority := 249, io_mode := 'R' );
    { Runs ,connects to SYAAIB , lowers
      its priority to STDPRI0 and
      then waits on SYAARQ }

    drvll_io( name := 'YAAWRT', priority := 249, io_mode := 'W' );
    { Runs, connects to SYAAIA ,lowers
      its priority to STDPRI0 and
      then waits on SYAAWQ }

    csr^.intenb := true;           {Enable (REQB) input interrupts}
    csr^.intena := true;           {Enable (REQA)output interrupts}
    change_priority( stdprior );
    {Lowers the Dispatcher priority}
END;

WHILE NOT stop
DO          {Head of request processing loop}
BEGIN

    get_packet( name := 'SYAA  ' ,
                packet_ptr := req_ptr::queue_ptr);
    {Process blocks if no requests queued}

    CASE req_ptr^.oper OF
        read_physical,
        read_logical:
            put_packet( name := 'SYAARQ', packet_ptr := req_ptr::queue_ptr );

        write_physical,
        write_logical:
            put_packet( name := 'SYAAWQ', packet_ptr := req_ptr::queue_ptr );

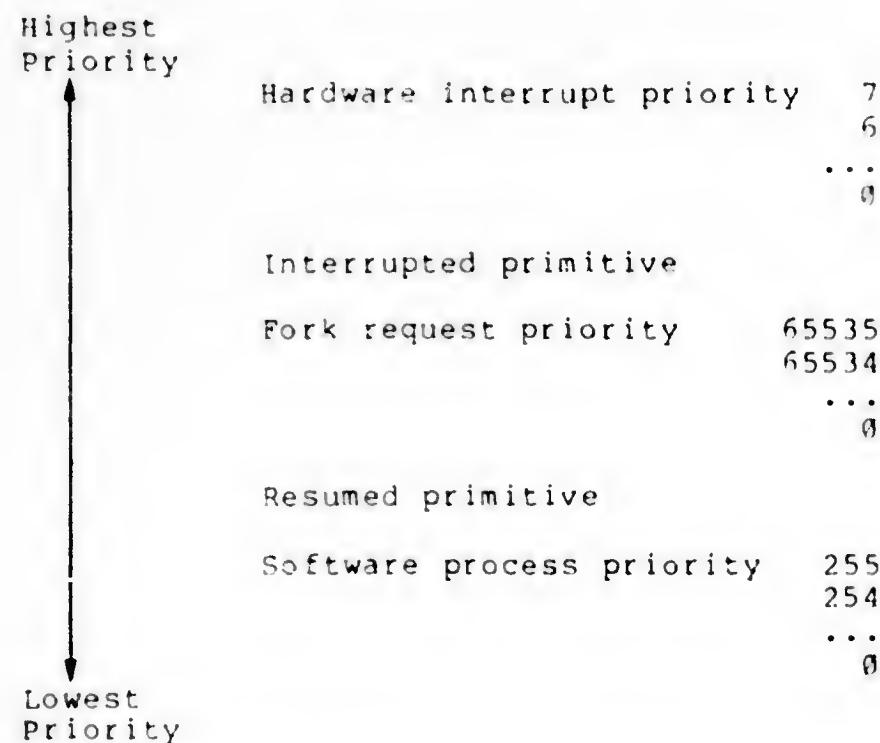
    get_characteristics:
    BEGIN
        req_ptr^.class:=DCSRLT;
        req_ptr^.typ:=RTSDRV;
        req_ptr^.subtype:=RS$PRT;
        reply_to_requestor
            (req_ptr,is$normal,io$success);
    END;
    OTHERWISE
        reply_to_requestor( req_ptr, is$function_inv, io$fatal );
        {ISSFUNCTION_INV = Invalid function for this driver}
        {IO$FATAL = severity level 5; placed in l.o. 3 bits of STATUS}
    END;
    {case REQ_PTR^.OPER}
END;
{End of request processing loop}

{ Here if error in initialization or STOP request }
destroy( name := 'SYAA  ' );
destroy( name := 'SYAARQ' );
destroy( name := 'SYAAWQ' );
destroy( name := 'SYAAIA' );
destroy( name := 'SYAAIB' );
{ Falling through here with no dependent processes will
  terminate the Dispatcher process and recovers its PCB memory?? }
END.
```

APPENDIX B  
SCHEDULING HIERARCHY AND RECOMMENDED PROCESS PRIORITIES

B.1 PRIORITY SCHEDULING HIERARCHY

MicroPower/Pascal implements the following priority scheduling hierarchy:



In the figure above, "interrupted primitive" refers to a primitive operation that was executing when a hardware interrupt occurred and thus did not complete. The interrupted operation is completed after all interrupts have been serviced but before any fork-level processing occurs and/or before the return to process level. "Resumed primitive" refers to a user-requested primitive operation that blocked part-way through execution, in a consistent state, because a resource was not available. When the resource becomes available, the continuation of that primitive operation is scheduled at the level shown above.

When an interrupt service routine (ISR) finishes its high-priority, interrupt-level processing, it exits by either dismissing the hardware interrupt or issuing a FORKS call. Scheduling then proceeds as follows: First, all pending interrupts and lower-level interrupted ISRs are processed. After all interrupts have been serviced, the kernel checks to see whether a primitive operation was interrupted; if so, it is allowed to complete.

## SCHEDULING HIERARCHY AND RECOMMENDED PROCESS PRIORITIES

Next, the kernel processes the priority-ordered fork request queue. After all fork requests have been serviced, the kernel checks to see whether any blocked primitive operations have unblocked; if so, they are allowed to complete.

Finally, the kernel returns to normal process-level scheduling, executing the highest-priority process in the ready-active state.

For more information on interrupt and fork processing, see Chapter 8 and the descriptions of the CINT\$, FORKS, and P7SYSS kernel requests in Chapter 3. For more information on software process priorities and process scheduling, see Chapter 2, the descriptions of the CHGPs\$, CRPC\$, DFSPCS, and SCHDS kernel requests in Chapter 3, and the rest of this appendix.

### B.2 RECOMMENDED PROCESS PRIORITIES

#### NOTE

In the current version of MicroPower/Pascal, all initialization procedures in Pascal processes execute at software process priority 255. The initialization priorities listed in this section are currently followed only by DIGITAL-supplied device handlers written in MACRO-11. This may change in a future release.

Recommended Priority Range	Process Type
0	Null process (lowest priority)
1-127	Least critical processes
128-159	Critical real-time processes
160-223	Device handlers
224-247	Most critical processes (ones that create structures that other processes can access)
224	Error-recording process
248-255	Initialization processes
248-249	(Reserved by DIGITAL)
250	Device handler initialization
251	Clock process initialization
252-253	(Reserved by DIGITAL)
254	Most critical process initialization
255	Error-recording process initialization (highest priority)

## SCHEDULING HIERARCHY AND RECOMMENDED PROCESS PRIORITIES

### B.3 RECOMMENDED DEVICE-HANDLER PROCESS PRIORITIES

#### NOTE

In the current version of MicroPower/Pascal, all initialization procedures in Pascal processes execute at software process priority 255. The initialization priorities listed in this section are currently followed only by DIGITAL-supplied device handlers written in MACRO-II. This may change in a future release.

Hardware Interrupt Request Level	Controller Process Priority Range	Recommended Initialization Process Priority Range
4	64-175	248-255 (250 recommended)
5	176-191	248-255 (250 recommended)
6	192-217	248-255 (250 recommended)
7	208-223	248-255 (250 recommended)

APPENDIX C  
KERNEL PRIMITIVES

A table of kernel primitive names, macro calls, and Pascal equivalents appears on the next page. In addition to kernel primitives, the table includes the Define Static Process, Fork Processing, Define an Impure Program Data Section, Define a Pure Program Data Section, Define a Pure Program Instruction Section, and Drop CPU Priority requests.

## KERNEL PRIMITIVES

Kernel Primitive Name	Macro Call	Pascal Equivalent
Conditionally Allocate Packet	A PCS area,qelm	COND ALLOCATE_PACKET function
Allocate Packet	ALPKS area,qelm	ALLOCATE_PACKET procedure
Connect to Exception Condition	CCNDS area,mask,group,sdb	CONNECT_EXCEPTION procedure
Change Process Priority	CHGPs area,pri	CHANGE_PRIORITY
Connect to Interrupt	CINT\$ area,vec,ps,val,imp,ist,pic	CONNECT_INTERRUPT procedure
Create Process	CPPCS area,pdb,pri,cxo,grp,ter,cxl,st,i, stl,sth,start,ini	Process-invocation statement
Create Structure	CRSTS area,sdb,styp,satt,value	CREATE_BINARY_SEMAPHORE function
Deallocate Packet	DAPKS area,qelm	DEALLOCATE_PACKET procedure
Dismiss Exception Condition	DEXCS area,pdb,action	RELEASE_EXCEPTION procedure
Define Static Process	DFSPCS pid,pri,typ,cxo,grp,ter,cxl,st,i, stl,sth,start,ini	Declaration of a [SYSTEM] PROGRAM
Disconnect from Interrupt	DINT\$ area,vec	DISCONNECT_INTERRUPT procedure
Delete Process	DLPC\$	(None)
Delete Structure	DLSTS area,sdb	DESTROY procedure
Fork Processing	FORK\$	(None)
Conditional Get Element	GELCS area,sdb,bufptr,bufcnt	COND_GET_ELEMENT function
Get Element	GELMS area,sdb,bufptr,bufcnt	GET_ELEMENT procedure
Get Process Status	GTSTS area,pdb,buf	GET_STATUS procedure
Return Structure Value	GVALS area,sdb,type,val	GET_VALUE procedure
Define an Impure Program Data Section	IMPURS	(None)
Define a Pure Program Data Section	PDATS	(None)
Conditional Put Element	PELCS area,sdb,bufptr,bufcnt	COND_PUT_ELEMENT function
Put Element	PELMS area,sdb,bufptr,bufcnt	PUT_ELEMENT procedure
Define a Pure Program Instruction Section	PURE\$	(None)
Drop CPU Priority	P7SYSS pri	(None)
Reset Ring Buffer	RBUFS area,sdb	RESET_RING_BUFFER procedure
Conditional Receive Data	RCVCS area,sdb,rtnptr,vlen,vbuf,rlen,rbuf	COND_RECEIVE function
Receive Data	RCVDS area,sdb,rtnptr,vlen,vbuf,rlen,rbuf	RECEIVE procedure
Report Exception	REXCS area,mask,subcod,arglen,argbuf	REPORT procedure
Resume Process	RSUMS area,pdb	RESUME procedure
Signal All Waiters	SALL\$ area,sdb	SIGNAL_ALL procedure
Schedule Process	SCHDS	SCHEDULE procedure
Send Data	SEND\$ area,sdb,pri,vlen,vbuf,rlen,rbuf	SEND procedure
Set Exception Routine Address	SERAS area,adr,mask	ESTABLISH procedure
Conditionally Signal Semaphore	SGLCS area,sdb	COND_SIGNAL function
Signal Queue Semaphore	SGLQS area,sdb,qelm	PUT_PACKET procedure
Signal Semaphore	SGNL\$ area,sdb	SIGNAL procedure
Conditionally Signal Queue Semaphore	SGQCS area,sdb,qelm	COND_PUT_PACKET function
Conditional Send Data	SNDCS area,sdb,pri,vlen,vbuf,rlen,rbuf	COND_SEND function
Suspend Process	SPND\$ area,pdb	SUSPEND procedure
Stop Process	STPC\$ area,pdb	STOP procedure
Conditionally Wait on Semaphore	WAICS area,sdb	COND_WAIT function
Wait on Queue Semaphore	WAIQS area,sdb,qelm	GET_PACKET procedure
Wait on Semaphore	WAIT\$ area,sdb	WAIT procedure
Conditional Wait on Queue Semaphore	WAQCS area,sdb,qelm	COND_GET_PACKET function

## APPENDIX D

### MACRO-11 SUBROUTINE CALLING CONVENTIONS

Subroutines written in MACRO-11 can be executed as procedures in a MicroPower/Pascal program. This appendix describes the conventions you must observe in order to invoke MACRO-11 subroutines from a MicroPower/Pascal program.

A MACRO-11 subroutine can be invoked from a MicroPower/Pascal program in two ways. The first is to use the normal MicroPower/Pascal subroutine calling sequence. The second is to use the SEQ11 directive (described in Chapter 6 of the MicroPower/Pascal Language Guide) to generate the standard PDP-11 subroutine calling sequence. Section D.1 describes the conventions associated with normal MicroPower/Pascal subroutine calls, and Section D.2 describes the conventions associated with subroutine calls that are generated with the SEQ11 directive.

#### D.1 NORMAL MICROPOWER/PASCAL SUBROUTINE CALLING CONVENTIONS

For a normal MicroPower/Pascal subroutine call, the parameters specified in the subroutine's procedure declaration are passed to the subroutine on the stack. The MicroPower/Pascal compiler pushes one actual parameter onto the stack for each formal parameter in the procedure declaration. Actual parameters are pushed onto the stack in the order in which the formal parameters were declared.

For VAR parameters, the address (one word) of the parameter is pushed.

For value parameters, the value of the actual parameter is pushed. Each value parameter pushed occupies an integral number of consecutive words of stack. The number of words occupied by a value parameter is dictated by the associated type (see Appendix F of the MicroPower/Pascal Language Guide). A minimum of one word is occupied by each value parameter.

For function invocations, the caller must allocate stack space for the returned function value. This space is an integral number of consecutive words, one word minimum. The number of words is dictated by the function result type (see Appendix F of the MicroPower/Pascal Language Guide). Stack space for the function return value must be allocated before any actual parameters are pushed onto the stack.

If the formal parameter list contains a procedure or function parameter, the associated actual parameter consists of two words. The first is a static link (zero for external subprograms, but normally points to the stack frame of the subprogram that most closely contains the actual subprogram). The second is the address of the actual procedure or function.

## MACRO-11 SUBROUTINE CALLING CONVENTIONS

The called procedure or function is responsible for removing the pushed actual parameters (if any) from the stack before returning, via an RTS PC instruction, to the caller.

Before returning from a function, the function value must be loaded into the return value stack slot, which was previously allocated by the caller immediately before pushing any actual parameters.

The sample Pascal program below illustrates the procedure declaration for a MACRO-11 subroutine (ADD) that is to be called with the normal MicroPower/Pascal calling sequence. Included in the procedure declaration are three parameters that will be passed to the ADD subroutine. The MACRO-11 subroutine code is shown following the Pascal program segment.

```
[SYSTEM(MICROPOWER), DATASPACE (300), STACKSIZE (100), PRIORITY  
 (10)]  
PROGRAM EXAMPLE;  
VAR  
    I,J,K : INTEGER;  
  
[EXTERNAL] PROCEDURE Add ( Addend_1, Addend_2 : INTEGER;  
                          VAR Sum : INTEGER) ;  
EXTERNAL;  
  
BEGIN  
    READ (I,J);  
    Add (I,J,K);  
    WRITE (K);  
END.
```

### MACRO-11 subroutine:

```
ADD::   MOV     R0,-(SP)      ; Save needed register  
        MOV     8(SP),R0      ; Initialize sum to first addend  
        ADD     6(SP),R0      ; Add second addend  
        MOV     R0,@4(SP)     ; Return the sum  
        MOV     (SP)+,R0       ; Restore used register  
        MOV     (SP)+,4(SP)    ; Move the return PC  
        CMP     (SP)+,(SP)+    ; Clean the stack  
        RTS     PC            ; Return
```

When the subroutine is entered, the state of the stack is as follows:

High address	Value of I	Value of J	Address of K	Return PC	SP
Low address					

Upon entry, the MACRO-11 subroutine saves any general registers it needs during execution. In this example, only R0 is saved. In the instructions that follow, operands are obtained via indexed addressing, with R0 accumulating the sum. Once the sum has been obtained, it is moved (via index-deferred addressing) to the address specified on the stack, which is the address of the variable K. Finally, the subroutine restores R0, cleans the stack, and returns to the calling program via an RTS PC instruction.

## MACRO-11 SUBROUTINE CALLING CONVENTIONS

Upon returning to the Pascal program segment, the variable K contains the sum of I and J. The Pascal program then writes the integer sum obtained via the MACRO-11 subroutine and ends.

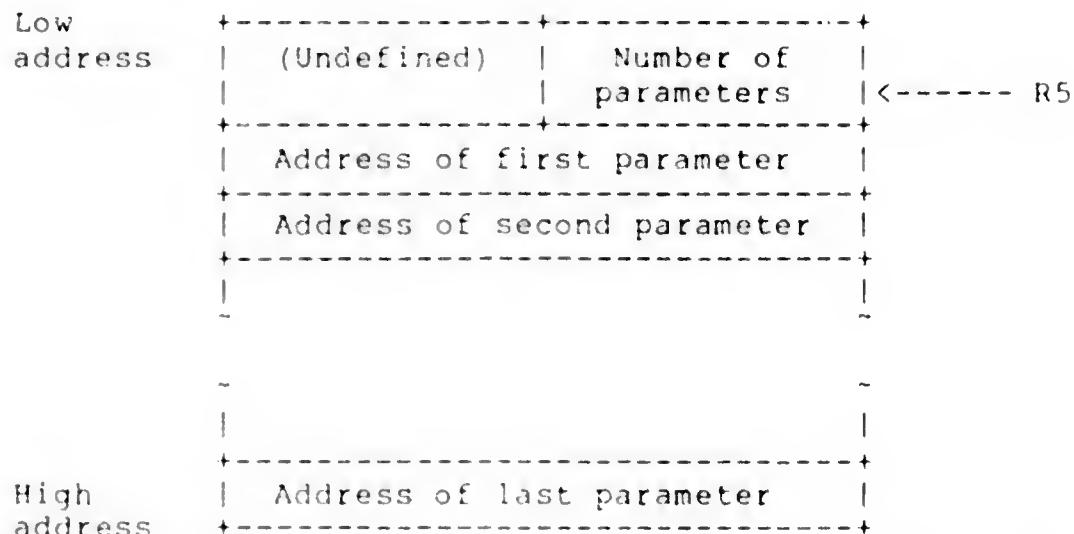
### D.2 STANDARD PDP-11 (SEQ11) SUBROUTINE CALLING CONVENTIONS

A standard PDP-11 subroutine call is generated by placing the SEQ11 directive immediately after the procedure declaration for the MACRO-11 subroutine to be called. (The SEQ11 directive is described in Chapter 6 of the MicroPower/Pascal Language Guide.) The SEQ11 directive causes the compiler to generate a call to the OTS. When executed at runtime, this call causes the OTS to generate (and initiate) the standard PDP-11 calling sequence.

#### NOTE

Since standard PDP-11 (SEQ11) subroutine calls are generated at runtime (and not at compile-time, like normal MicroPower/Pascal subroutine calls), heavy use of standard PDP-11 calls may degrade system performance.

For a standard PDP-11 subroutine call, the parameters specified in the subroutine procedure declaration are passed to the subroutine via a parameter list. When the subroutine is entered, R5 contains the address of the parameter list, which the OTS has constructed as follows:



Upon returning from the subroutine, the OTS cleans the stack and returns to the Pascal program.

The sample Pascal program below illustrates the procedure declaration for a MACRO-11 subroutine (ADD) that is to be called with the standard PDP-11 subroutine calling sequence. Included in the procedure declaration are three parameters that will be passed to the ADD subroutine. Immediately following the procedure declaration is the SEQ11 directive. The MACRO-11 subroutine code is shown following the Pascal program segment.

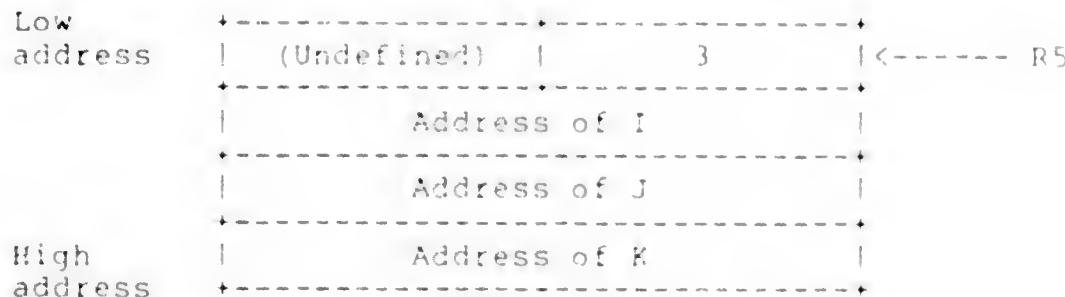
## MACRO-11 SUBROUTINE CALLING CONVENTIONS

```
[SYSTEM(MICROPOWER), DATASPACE (300), STACKSIZE (100), PRIORITY  
 (10)]  
 PROGRAM EXAMPLE;  
 VAR  
   I,J,K : INTEGER;  
  
 [EXTERNAL] PROCEDURE Add ( VAR Addend 1,  
                           Addend 2,  
                           Sum : INTEGER) ;  
 SEQ1;  
  
 BEGIN  
   READ (I,J);  
   Add (I,J,K);  
   WRITE (K);  
 END.
```

### MACRO-11 Subroutine:

```
ADD::  MOV    R0,-(SP)      ; Save needed register  
       TST    (R5)+          ; Point to first addend  
       MOV    @ (R5)+,R0      ; Initialize sum to first addend  
       ADD    @ (R5)+,R0      ; Add second addend  
       MOV    R0,@ (R5)        ; Return the sum  
       MOV    (SP)+,R0         ; Restore R0  
       RTS    PC              ; Exit to OTS
```

The parameter list passed to the MACRO-11 subroutine via R5 is as follows:



Upon entry, the MACRO-11 subroutine saves any general registers it needs during execution. In this example, only R0 is saved. The TST instruction increments the parameter list pointer (R5) so that it points to the second word in the parameter list, which contains the address of the variable I. In the instructions that follow, operands are obtained via R5 auto-increment-deferred addressing, with R0 accumulating the sum. Once the sum has been obtained, it is moved to the third address specified in the parameter list, which is the address of the variable K. Finally, the subroutine restores R0 and returns to the OTS via an RTS PC instruction.

Upon returning to the Pascal program segment, the variable K contains the sum of I and J. The Pascal program then writes the integer sum obtained via the MACRO-11 subroutine and ends.

## APPENDIX E

### MICROPOWER/PASCAL 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. Disks, diskettes, and TU58 cassettes are directory-structured devices.

The directory segments contain entries describing the names, lengths, and creation dates of files on the device. Because the directory is at the beginning of the device, you can access directly any file on the device, regardless of its location. Therefore, directory-structured devices are sometimes called random-access or block-replaceable devices.

MicroPower/Pascal software includes a directory service process, named RTDSP, for creating and maintaining directories. RTDSP 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. Next, the appendix describes the contents of a device directory and explains how to recover information from a random-access device directory that is corrupted. Then the appendix discusses file storage.

#### E.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 E-1 shows the format of a random-access device.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

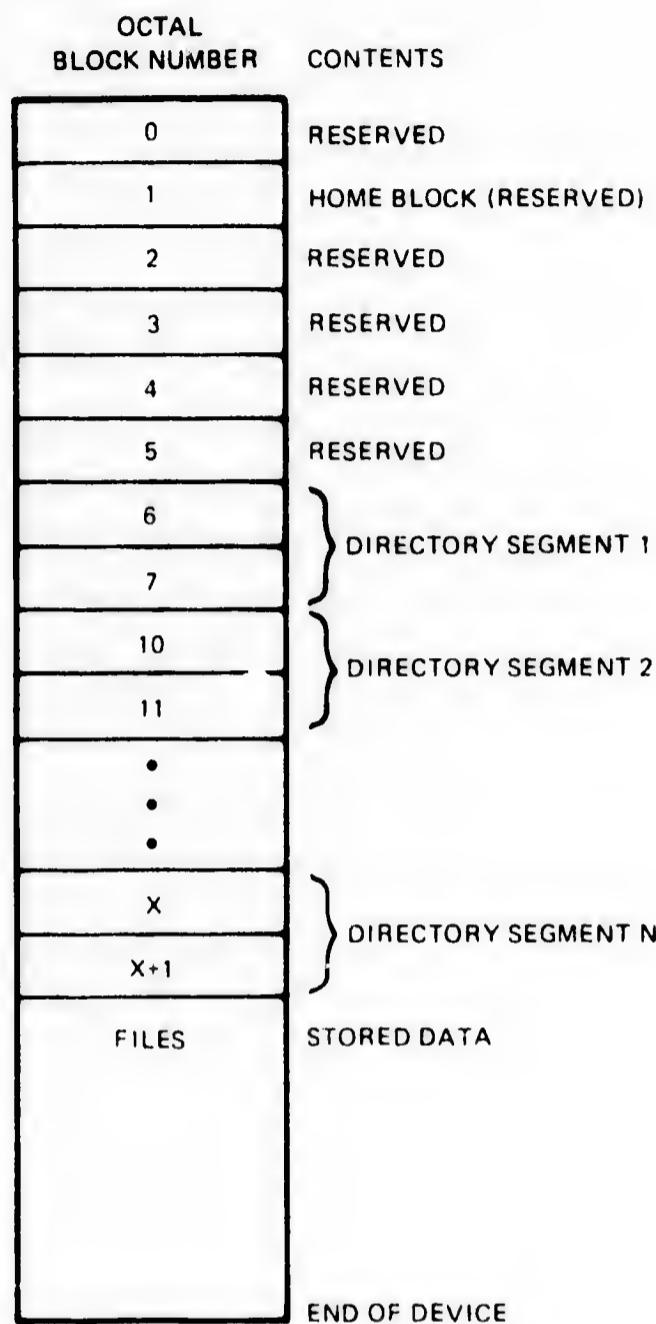


Figure E-1 Random-Access Device

### E.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 E-2 and Table E-1 show the format and contents of the home block.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7					
000	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a					
040	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a					
100	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a					
140	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a					
200	a	a	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b					
240	b	b	b	b	b	b	b	b																													
300																																					
340																																					
400																																					
440																																					
500																																					
540																																					
600																																					
640																																					
700	c	c	d	d																e	e	f	f	g	g	h	h	h	h	h	h	h	h	h	h	h	
740	h	h	h	h	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	k	k				

Figure E-2 Format of Home Block

Table E-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	

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

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.

### E.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(decimal) 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 header and entries containing file information. Each segment ends with an end-of-segment marker. Figure E-3 shows the general format of the device directory.

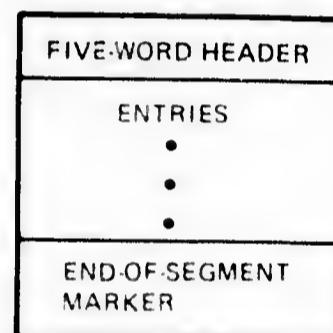


Figure E-3 Format of Device Directory

**E.1.2.1 Directory Header** - The directory segment header consists of five words; the remaining 507 words of the two-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.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

- 3 The number of the highest segment in use. RTDSP 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 (see Section E.1.2.2).
- 5 The block number on the device at which the stored data monitored by this segment begins.

E.1.2.2 Directory Entry - The rest of the directory segment consists of directory entries, followed by an end-of-segment marker. Figure E-4 shows the format of a directory entry.

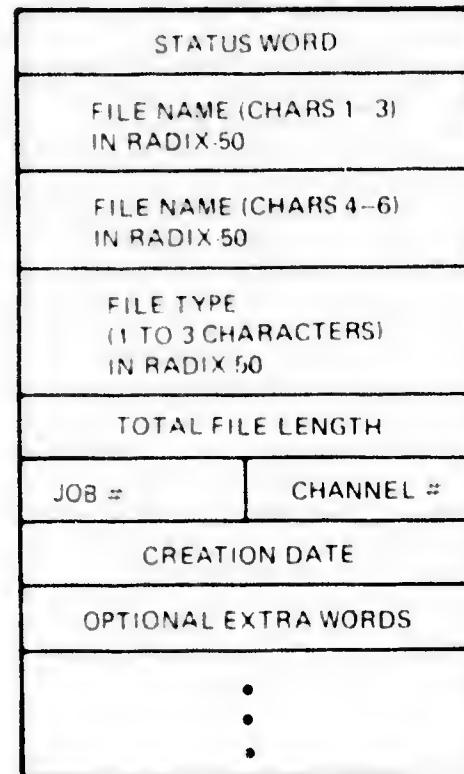


Figure E-4 Format of Directory Entry

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 E-5 illustrates the status word.



Figure E-5 Format of Status Word

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

There are five valid status word values, as follows:

Status Word	Meaning
400	Tentative file.
1000	Empty area (RTDSP 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. RTDSP uses this marker 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 name, file type, or other entry information.

RTDSP 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, RTDSP 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. The RT-11 DIR utility program lists tentative files that appear in directories as <UNUSED> files.
- An empty entry defines an area of the device that is available for use. Thus, when you delete a file, you obtain an empty area. DIR lists an empty area as <UNUSED>, followed by the length of that 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, RTDSP deletes the first file, thus replacing the old file with the new file. DIR lists permanent files that appear in directories by their file names, file types, sizes, and creation dates.

### NOTE

RTDSP 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 or the IFS\$PRO and IFS\$UNP functions of RTDSP (see Chapter 5). For more information, see the RT-11 System User's Guide.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

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, RTDSP normally ignores these words. However, the RT-11 DIR command with the /DELETED option lists the names of deleted files.

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 RTDSP.

The seventh word of a directory entry contains the file's creation date. When a program creates a tentative file, RTDSP moves the system date word into the creation date slot for the entry. To have your files dated, you must include the DIGITAL-supplied clock process (see Chapter 6). If there is no clock process, or if no date was set, the date word equals 0. Figure E-6 shows the format of the date word. Bit 15 is used to indicate the end-of-file status.

15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
	MONTH IN DECIMAL (1-12)	DAY IN DECIMAL (1-31)	YEAR MINUS 110 IN OCTAL

Figure E-6 Format of Date Word

Normally, directory entries are seven words long, but by using the DUP utility program with the /Z:n option, you can allocate extra words for each entry when you initialize the device. The fourth word of the directory header contains the number of extra bytes you specify. Although the RT-11 DUP command lets you allocate extra words, RTDSP provides no means to manipulate this extra information conveniently. When RTDSP 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.

### E.2 DIRECTORY USE

#### E.2.1 Sample Directory Segment

The directory listings shown in Figure E-7 describe a double-density diskette with 11 files.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

### DIRECTORY/FULL BY0:

29-APR-79						
SWAP <sup>1</sup> .SYS	24	19-FEB-79	RTFSJ.SYS	65	19-FEB-79	
UNUSED	77		PFP <sup>1</sup> .SAV	16	19-FEB-79	
DPF <sup>1</sup> .SAV	21	19-FEB-79	PR <sup>1</sup> .SAV	17	19-FEB-79	
EDIT <sup>1</sup> .SAV	19	19-FEB-79	LNN <sup>1</sup> .SAV	32	19-FEB-79	
LBR <sup>1</sup> .SAV	20	19-FEB-79	BUMP <sup>1</sup> .SAV	1	19-FEB-79	
MACRO <sup>1</sup> .SAV	45	19-FEB-79	STP <sup>1</sup> .SAV	13	19-APR-79	
UNUSED	-	613				
11 FILES, 284 BLOCKS						
690 FREE BLOCKS						

### DIRECTORY SUMMARY BY0:

29-APR-79						
11 FILES IN SEGMENT 1						
4 AVAILABLE SEGMENTS, 1 IN USE						
11 FILES, 284 BLOCKS						
690 FREE BLOCKS						

Figure E-7 Directory Listings

Figure E-8 shows the contents of segment 1 of the diskette directory, obtained by dumping absolute block number 6 of the device.

HEADER:	4 0 1 0 16	FOUR SEGMENTS AVAILABLE NO NEXT SEGMENT HIGHEST OPEN IS #1 NO EXTRA BYTES PER ENTRY FILES START AT DEVICE BLOCK 16 OCTAL
ENTRIES:	2000 75131 62000 75273 30 - 5147	PERMANENT FILE RADIX-50 FOR SWA RADIX-50 FOR P RADIX-50 FOR SYS FILE IS 30 OCTAL BLOCKS LONG USED ONLY FOR TENTATIVE FILES CREATED ON 19-FEB-79
	2000 71677 142302 75273 101 - 5147	PERMANENT FILE RADIX-50 FOR RT1 RADIX-50 FOR 1SJ RADIX 50 FOR SYS 101 OCTAL BLOCKS LONG USED ONLY FOR TENTATIVE FILES CREATED ON 19 FEB 79

1000 16315 54162 75273 115 - 5147	EMPTY AREA (THE FILE DXMNFB WAS DELETED) RADIX 50 FOR DXM RADIX 50 FOR NFB RADIX-50 FOR SYS 115 OCTAL BLOCKS LONG USED ONLY FOR TENTATIVE FILES CREATED 19 FEB 79
---	---

(continued on next page)

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

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
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 RD
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

Figure E-8 Directory Segment

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

To find the starting block of a particular file, first find the directory segment containing the entry for that file. Next, in the fifth word of that directory segment, add to the starting block number the length of each permanent, tentative, and empty entry in the directory before your file. In Figure E-8, for example, the permanent file RT11SJ.SYS begins at block number 46(octal) on the device.

### E.2.2 Splitting a Directory Segment

Whenever RTDSP 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 E-9 shows how RTDSP stores a new file as a tentative file followed by an empty area.

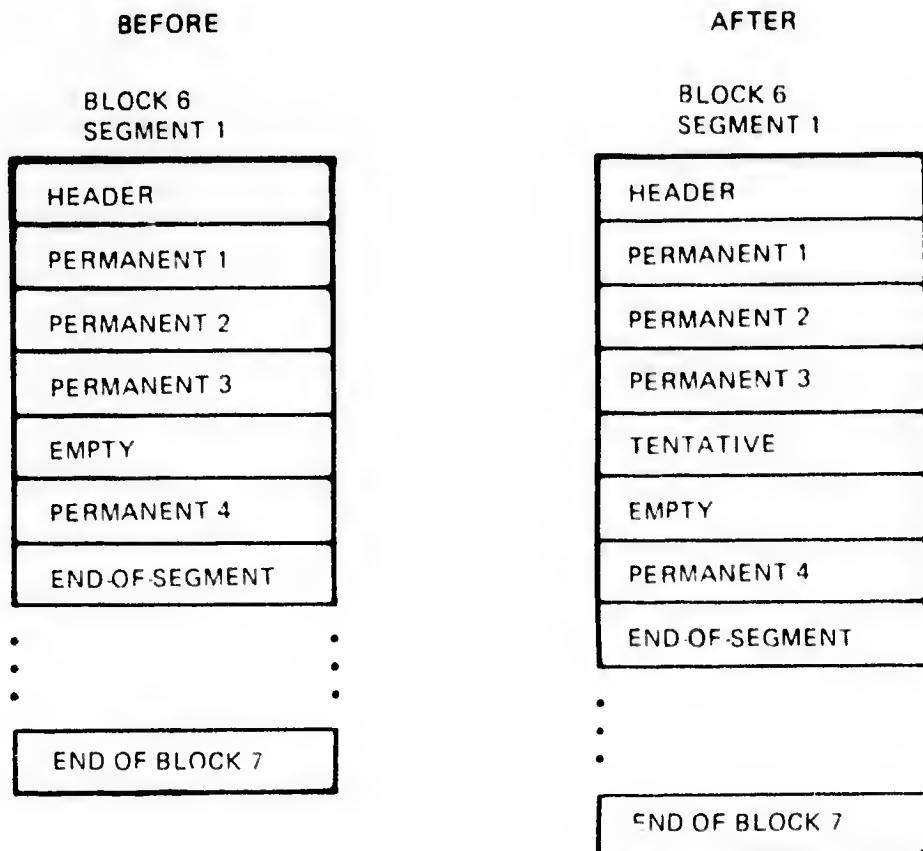


Figure E-9 Storing a New File

This procedure works properly if the empty entry and the entries following it can move downward. However, if the segment is full, RTDSP must split the segment, if possible, in order to store the new entry. Figure E-10 illustrates a directory segment that is full.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

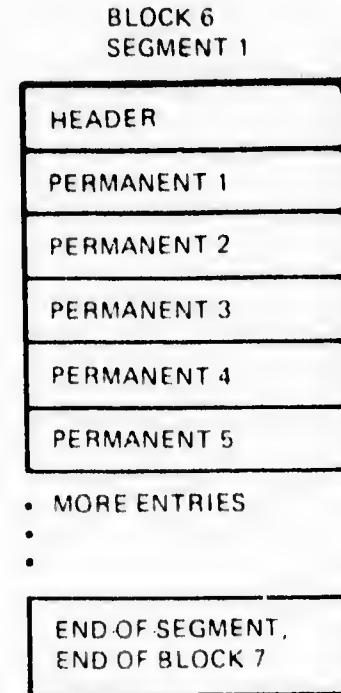


Figure E-10 Full Directory Segment

First, RTDSP checks the header for the number of segments available. If there are none, a "directory full" error results, and you cannot store the new file. You can squeeze the volume at this point by using RT-11 commands to pack the directory segments and can try the operation again. You could also delete any unneeded files.

If another directory segment is available, RTDSP 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, RTDSP 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, RTDSP restores the first word of the middle entry to the copy of the segment that is still in memory and restores the link information. RTDSP slides the middle entry and all subsequent entries to the top of the segment. Then RTDSP writes this segment to the volume as the next available segment. Finally, RTDSP reads segment 1 into memory and updates the information in its header, at which point RTDSP begins its search again for a suitable empty entry to accommodate the new file.

Figures E-11 and E-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.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

HIGHEST SEGMENT IN USE: 1  
NUMBER OF SEGMENTS AVAILABLE: 2

BLOCK 6  
SEGMENT 1

HEADER
PERMANENT 1
PERMANENT 2
PERMANENT 3
PERMANENT 4
PERMANENT 5
EMPTY
PERMANENT 6
PERMANENT 7
END-OF SEGMENT, END OF BLOCK 7

BLOCK 10  
SEGMENT 2

END OF BLOCK 11

Figure E-11 Directory Before Splitting

HIGHEST SEGMENT IN USE: 2  
NUMBER OF SEGMENTS AVAILABLE: 2

BLOCK 6  
SEGMENT 1

HEADER
PERMANENT 1
PERMANENT 2
PERMANENT 3
PERMANENT 4
END-OF-SEGMENT
•
•
•
END OF BLOCK 7

BLOCK 10  
SEGMENT 10

LINK	→	HEADER
		PERMANENT 5
		EMPTY
		PERMANENT 6
		PERMANENT 7
		END-OF-SEGMENT
		•
		•
		•
		END OF BLOCK 11

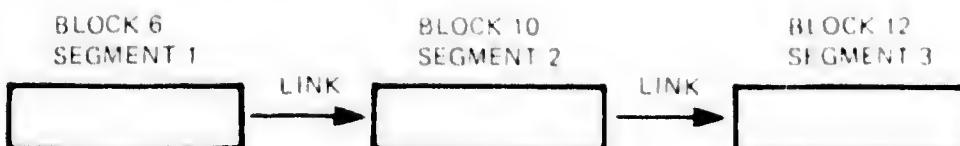
Figure E-12 Directory After Splitting

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

After a directory segment splits, RTDSP 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 is the empty area in Figure E-12.

Thus far, the link words seem 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; 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 E-13 illustrates this example.

HIGHEST SEGMENT IN USE 3  
NUMBER OF SEGMENTS AVAILABLE 4



HIGHEST SEGMENT IN USE 4  
NUMBER OF SEGMENTS AVAILABLE 4

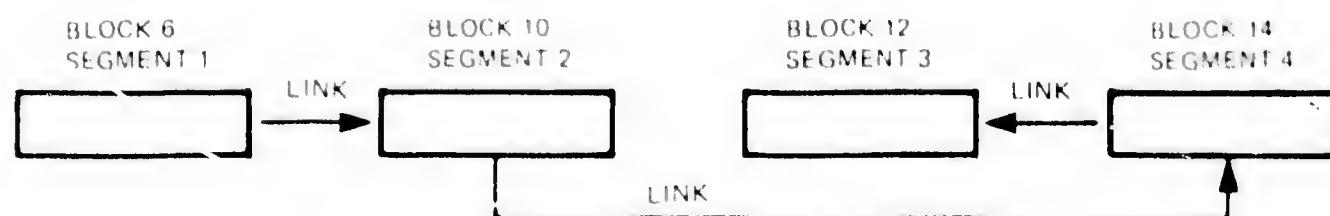


Figure E-13 Directory Links

### E.2.3 Using RT-11 Commands to Recover Data from a Corrupted Directory

One of the most frustrating experiences you can have as a programmer is to lose data on a volume because a block in the device directory went bad or because another user wrote over the directory. Usually, in a situation like this the files on the volume are intact, but the directory entries for some of the files have been destroyed. This section presents some guidelines for recovering as much data as possible from a volume with a corrupted directory.

**E.2.3.1 Examine Segment 1** - Your first step in recovering data is to determine whether segment 1 of the directory is bad. Remember, segment 1 occupies physical blocks 6 and 7 of the device. To examine segment 1, mount the volume and try to get an ordinary listing of the files. Use the RT-11 DIR command without any options.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

An immediate ?MON-F-Directory I/O error or ?MON-F-Dir I/O error message indicates that segment 1 is bad. You now have two options:

1. Reformat and reinitialize the volume (it is reusable if a bad-block scan shows no bad block in the directory area).
2. Open the volume in non-file-structured mode with TECO and search for data that resembles source code or other ASCII information that looks familiar.

Since this kind of search is a tedious process, however, you probably should not even consider it unless you have a video terminal to use with TECO. See Section E.2.3.3 for information on removing a file from the volume.

If, on the other hand, the RT-11 DIRECTORY command gives you at least a partial directory listing, you will be able to recover some of the information from the volume by issuing the RT-11 DIR/S command. The /S option lists information up to but not including the bad segment. To recover as many files as possible, you must repair the directory by linking around the bad segment.

**E.2.3.2 Follow the Chain of Segments** - Use the RT-11 SIPP utility to open the volume in non-file-structured mode. Look first at location 6000--the start of the header for directory segment 1. It contains the total number of segments available. Location 6002 contains the number of the next segment, and location 6004 shows the highest segment in use (see Section 1.2.1).

To find the absolute location of the next segment, multiply the link word by 2000 and add 4000. For example, if the link word is 2, the next segment starts at location 10000 on the volume. Chain your way through the segments by opening the next segment and following its link word. As you go, make a worksheet for the link information, according to the format shown in Figure E-14. Continue chaining until you have accounted for all the segments. Remember that segment 1 is always the first segment -- that is, nothing links to it. The last segment always links to 0.

HIGHEST SEGMENT IN USE *	
NUMBER OF SEGMENTS AVAILABLE *	
SEGMENT	LINKED TO
1	2
2	4
4	3(bad)
3(bad)	?

Figure E-14 Worksheet for a Directory Chain with Four Segments

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

In Figure E-14, segment 3 must link to 0 -- the last segment -- since all the others have been accounted for already. To repair this directory, modify the header of segment 4 so that the link word contains 0 instead of 3. This eliminates segment 3 from the chain. Section E.2.3.3 describes how to remove the files from the volume.

Figure E-15 shows a more complicated example. In this case, the bad segment is not the last one in the directory.

HIGHEST SEGMENT IN USE: 9  
NUMBER OF SEGMENTS AVAILABLE: 9

SEGMENT: LINKED TO:

1	2
2	5
5	4
4	8
8	7(bad)
7(bad)	?
3	0
6	9
9	3

Figure E-15 Worksheet for a Directory Chain with Nine Segments

In a situation of the kind shown in Figure E-15, you can follow the chain from segment 1 through segment 8, which points to the bad segment. To continue from that point, enter in the left column the lowest segment number not yet accounted for and follow its link. Remember, if a segment links to 0, it is the last segment in the directory. Continue until all the segments are accounted for in the leftmost column. When you finish, the number missing from the rightmost column is the segment to which the bad segment links (in Figure E-15, segment 6). As in the previous example, use SIPP to link around the bad segment. In this case, change the link word in segment 8 to point to segment 6, thus removing segment 7 from the chain.

**E.2.3.3 Remove the Data from the Good Segments** - After eliminating the bad segment by linking around it, you are ready to save the files whose entries appear in the good segments. Use the RT-11 COPY command to copy the files to a good volume. The following command, for example, copies all the files from one diskette to another:

**COPY DY0:\*.\* DY1:**

This procedure removes all the files from the volume except those whose entries appear in the bad segment.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

**E.2.3.4 Remove the Data from the Bad Segment** - You can sometimes save files whose entries appear in the bad segment by using SIPP and DUP. If block 1 of the segment is unreadable when you open the segment with SIPP, you should probably give up; if block 2 of the segment is readable, it probably contains old, invalid data.

If you can read block 1, decode the header and the entries according to the diagram in Figure E-4. Continuing with SIPP, try to locate the files on the volume. (Section E.2.1 explains how to locate a file on the volume.) After establishing the starting and ending blocks of a specific file, run DUP and use the following command sequence to transfer to a new device:

```
output-filename=input-device:/G:startblock/E:endblock/I/F
```

You can use the following keyboard monitor command to achieve the same results:

```
COPY/DEVICE/FILE/START:startblock/END: endblock input-device  
output-filename
```

## E.3 FILE STORAGE

RTDSP 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.

### E.3.1 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 best suit typical MicroPower/Pascal applications.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

Figure E-16 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.

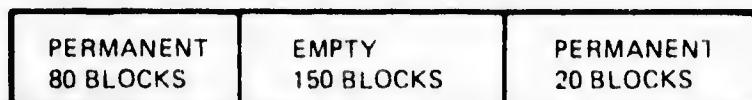


Figure E-16 Random-Access Device with Two Permanent Files

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.

RTDSP 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 RTDSP to recover unused space if less data is written to the file than you originally estimated. Figure E-17 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 E-17 Random-Access Device with One Tentative File

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 E-18. Remember that every tentative file must be followed by an empty area.

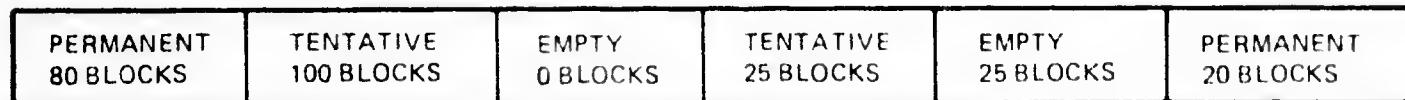


Figure E-18 Random-Access Device with Two Tentative Files

When RTDSP finishes writing data to the device, it closes the tentative file. RTDSP 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 E-19 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.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

PERMANENT 80 BLOCKS	PERMANENT 75 BLOCKS	EMPTY 25 BLOCKS	PERMANENT 10 BLOCKS	EMPTY 40 BLOCKS	PERMANENT 20 BLOCKS
------------------------	------------------------	--------------------	------------------------	--------------------	------------------------

Figure E-19 Random-Access Device with Four Permanent Files

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 the new data to the new file. You can then delete the old file.

The RT-11 DUP utility program provides the /T option as an easy way to extend the size of an existing file. However, to use this option, you must have an empty file with sufficient space in it immediately following the data file. (You can also access this option through the RT-11 CREATE/EXTENSION command.)

### E.3.2 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 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, an end-of-segment entry must exist. Note that on a disk squeezed by DUP, the end-of-segment entry might not be a full entry, but may contain just the status word.

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 RTDSP handles filled directory segments.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

When a directory segment becomes full, requiring the opening of a new segment, RTDSP 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 E.2.2 for a detailed explanation of how RTDSP splits a directory segment.

If you add files continually to a device without issuing the RT-11 SQUEEZE monitor command, you can use the following formula to compute the maximum number of entries and, thus, the maximum number of files:

$$S = \frac{(M - 1) * S}{2} + S$$

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 compressing the device by using the RT-11 SQUEEZE command when the directory fills up. DUP packs the directory segments as well as the physical device.

### E.4 RTDSP MESSAGE FORMAT

RTDSP, the directory service process, is a separate process in your application; RTDSP performs all operations involving device directories. RTDSP waits on the queue semaphore named \$RTDSP for a message packet that specifies the function to be performed, to which a semaphore reply should be sent, and several other parameters.

RTDSP uses the same message packet to reply to the requesting process, but returns some new information. Section E.4.2 describes replies.

#### E.4.1 Requests

Either \$QIO or a MACRO-11 user process can send a request to RTDSP by using the SENDSS primitive (see Chapter 3). For example:

```
SENDSS    sbd=#DSP, pri=R0, vlen=#FS.SSZ, vbuf=#MSG,  
          rlen=#FIBSIZ-6, rbuf=#FIB+2
```

Figure E-20 shows the format of the request message that RTDSP expects.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

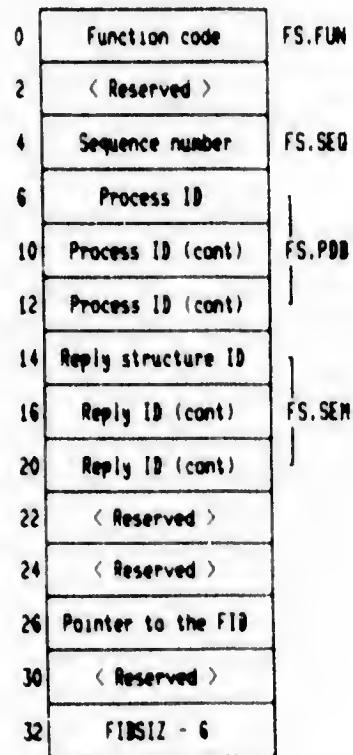


Figure E-20 RTDSP Request Message Format

The fields of the request semaphore message are defined as follows:

Field	Definition
FS.FUN	The function code for the operation requested from RTDSP. The legal function codes are:  IF\$DIR      IF\$PRO      IF\$UNP      IF\$PRG IF\$REN      IF\$DEL      IF\$INI      IF\$CLS IF\$LKP      IF\$ENT
FS.SEQ	The sequence number of the request.
FS.PDB	The structure ID of the structure descriptor block for the process requesting the operation.
FS.SEM	The structure ID of the structure descriptor block that the device driver will use to reply to RTDSP.
26(octal)	A pointer to the second word of the file identification block associated with the request.
32(octal)	The size of the argument pointed to in 26 (above). The value supplied here is FIBSIZ-6.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

### E.4.2 Replies

After RTDSP attempts to perform the operation requested, it replies using the reply semaphore specified in the request. The reply is made regardless of the success or failure of the attempt to perform the requested operation. If the attempt was successful, a value of 0 is returned in the status word; otherwise, RTDSP returns an error or an exception (see Chapter 4 of the MicroPower/Pascal Messages Manual).

To wait for a reply from RTDSP, SQIO or a MACRO-11 user process must wait on the reply semaphore using the RCVD\$S primitive (see Chapter 3). For example:

```
RCVD$S    sdb=#FB.REP(R0), rtnptr=#TEMP, vlen=#FS.RSZ,  
          vbuf=#msg, rlen=#0, rbuf=#0
```

Figure E-21 shows the format of the reply message that RTDSP uses.

0	Function code	FS.FUN
2	< Reserved >	
4	Sequence number	FS.SEQ
6	Reply status code	FS.STS
10	File status update	FS.ALU
12	< Reserved >	
14	File size	FS.FSZ
16	Starting block	FS.OFF
20	Starting block (cont)	
22	< Reserved >	
24	< Reserved >	
26	< Reserved >	
30	< Reserved >	
32	< Reserved >	

Figure E-21 RTDSP Reply Message Format

The fields of the reply semaphore message are defined as follows:

Field	Definition
FS.FUN	The function code for the operation requested. This field is returned unchanged to the requesting process.
FS.SEQ	The sequence number of the request. This field is returned unchanged to the requesting process.
FS.STS	The status code for the operation performed or attempted. When the SQIO receives a reply, this value is copied into the FB.CST field of the file identification block.

## MICROPOWER/PASCAL DIRECTORY STRUCTURE AND FILE STORAGE

- FS.ALN** An update to the status word (FB.STA) of an open file. This update may be the codes \$NEWFL, \$NFNSIO, \$NFSDV, or zero. When \$QIO receives a reply from RTDSP, \$QIO essentially ORs this value with the previous value of the FB.STA field to create a new value for FB.STA.
- FS.FSZ** The size of the opened file, in blocks. When \$QIO receives a reply from RTDSP, \$QIO copies this value into the FB.FSZ field of the file identification block in use.
- FS.OFF** The offset of the start of the opened file from the beginning of the device. Upon receiving a reply from RTDSP, \$QIO copies this value into the FB.SBK field of the file identification block in use.

APPENDIX F  
MACRO-11 FILE SYSTEM EXAMPLE

```
.nlist      ;Edit Level 1
.enabl LC
.list
.title MACIO - MACRO-11 DSP Interface Example Application
.ident /X01.01/
;+
;  
; Module name: MACIO.MAC
;  
; System: Micro-PAX/Pascal
;  
; Functional Description:
;  
; This is an example application program, written in MACRO-11, which
; uses the MicroPower/Pascal Version 1 file system. This program
; shows the definition and use of the file identification block (FIB),
; the QIO$ request, and how to open, read, write, and close files
; using the directory structure process (RTDSP).
;  
; Input to the program is an input file specification and an output
; file specification from a terminal. The program then reads the
; input file and writes it to the output file.
;-
.sbttl Declarations
.sbttl * System Macro Requirements
;+
;  
; System Macro requirements
;  
; MACDF$ - Define useful system macros
; IODFS - Define system I/O symbols
; QUEDFS - Define system queue symbols
; DRVDFS - Define device driver symbols (in FSINT$)
; EXMSKS - Define exception symbols
;-
Miscellaneous Primitives
;-

.enabl GBL

.mcall MACDF$, IODFS$, QUEDFS$, DRVDFS$, EXMSKS$

macdf$  
iodfs$  
quedfs$  
exmsk$  

.mcall CRST$$, ALPK$$, DAPK$$, DLST$$, GELM$$, PELM$$
.mcall SEND$$, GTST$$, RCVD$$, DLPC$, GVAL$$, STPC$$
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
;+
;
; Local macro definitions
;
;-
;
; SDB$ - Define the impure space for a structure descriptor block
;
        .MACRO  SDB$           ;+
        .blkb   SD.SIZ          ; The size of a descriptor
        .even
        .ENDM   SDB$           ;-
;
; FIB$ - Define impure space for FIB
;
        .MACRO  FIB$           ;+
        .blkb   FIBSIZ         ; size of FIB
        .even
        .ENDM   FIB$           ;-
;
; .PRINT - Types an ASCIIZ message to a terminal associated
;           with structure descriptor block named TTOSDB
;
        .MACRO  .PRINT  MSG
        MOV     MSG,R0           ; Get address
        CALL    STTOUT          ; Print the message
        .ENDM   .PRINT           ;-
;
; .GTLIN - Reads an ASCII line from the terminal associated
;           with the structure descriptor block named TTISDB
;
        .MACRO  .GTLIN  BUF
        MOV     BUF,R0           ; Get address of buffer
        CALL    STTIN           ; Read the input line
        .ENDM   .GTLIN           ;-
;
;+
;
; External symbols which are necessary to assemble this module but
; which may have default definitions - None.
;
;-
;
        .sbttl * Local Symbol Definitions
;
;+
;
; Data and symbols owned by this module are defined here and storage is
; allocated in the appropriate (pure or impure) data section.
;
;-
;
;+
;
; Define FIB offsets and related symbols
;
; FSINT$ - Define FIB symbols
; QIOS - File System message interface
; PARSE$ - File System file specification parser interface
;
;-
;
        .mcall   FSINT$, QIOS$, PARSE$           ;+
        fsint$                                     ;-
;
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
; Miscellaneous symbols

CR      = 15          ; ASCII return
LF      = 12          ; ASCII line-feed
DEL     = 177         ; ASCII rubout
BS      = 10          ; ASCII backspace
SPACE   = 40          ; ASCII space

IBFSZ   = 512.        ; Input file buffer size
OBFSZ   = 512.        ; Output file buffer size
MIOSSZ  = 200.        ; Stack size

; Define global message priority value

QIO$PR == 50.          ; Message priority for QIO routine
.sbt1 * Pure Data Definition

pdat$  

MIOPUR:  

    .nlist BEX
    IPROMP: .asciz /INPUT FILE? /
    OPROMP: .asciz /OUTPUT FILE? /
    EFNF:   .asciz /?File Not Found/<CR><LF>
    ECCO:   .asciz /?Can't Open Output File/<CR><LF>
    EIFS:   .asciz /?Improper File Specified/<CR><LF>
    ENSD:   .asciz /?No Such Device/<CR><LF>
    EERF:   .asciz /?Error Reading File/<CR><LF>
    EEWF:   .asciz /?Error Writing File/<CR><LF>
    EWPE:   .asciz /?Writing Past EOF/<CR><LF>
    LFCH:   .byte  LF ; Line-feed for <CR> echo
    BSCH:   .byte  BS,SPACE,BS      ; Rub out sequence
    LBSCH = . - BSCH
    .even
    .list BEX
    .sbttl * Impure Data Definition

impur$ ; Define impure data area

MIOIMP:  

    INFIL: fib$           ; Input file FIB
    OUTFIL: fib$           ; Output file FIB
    DSPSDB: sdb$           ; SDB for RTDSP requests
    RSDB:   sdb$           ; Request reply SDB
    TTISDB: sdb$           ; Terminal input ring buffer SDB
    TTOSDB: sdb$           ; Terminal output ring buffer SDB
    TMPSDB: sdb$           ; Work area to copy driver SDB
    LNBUF:  .blkw  40.      ; Area for GTST$, terminal input
    INBUF:  .blkb  IBFSZ    ; Input file buffer
    .even
    OUTBUF: .blkb  OBFSZ    ; Output file buffer
    .even

    MIOISZ = . - MIOIMP      ; Size of the impure area

    MIOSTK: .blkb  MIOSSZ    ; Room for stack
    .even
    MIOSTI:   .blkb           ; Initial stack start
    MIOSTH:   .blkw           ; End of Stack

    .sbttl * Static Process Definition

    MIOIPR = 50.            ; Process priority
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
; Define the static process

dfspc$ MIO,MIOIPR,PT.GEN,,,MIOEND,,MIOSTK,MIOSTH,MIOSTI,MIOINI

    .sbttl Start of Code
    .sbttl * MIOEND - Process termination

    pure$ ; This data is pure(can be put in ROM).

MACIO:::                                ; Global for the map

;+
; MIOEND - Process termination routine
;
; Executes when the process dies or is otherwise stopped.
; This code destroys the structures and deletes the process.
;-

MIOEND:
    TST      RSDB          ; Finished Initializing?
    BEQ      10$           ; No

    10$:   dlist$ #RSDB       ; Delete reply semaphore
    dlpcl$                         ; Delete process
    .sbttl * MIOINI - Process initialization

;+
; MIOINI - Process initialization
;
; Creates the structures to be used and sets up SDB's
;-
;

; This section of code clears the impure data area

MIOINI:
    MOV      #MIOIMP,R1        ; Move address into R1
    MOV      #MIOISZ/2,R0        ; size in words(bytes/2)
    1$:    CLR      (R1)+        ; Clear a word, increment,
    SOB      R0,1$             ; loop, until finished

; This section of code sets up the SDBs for terminal I/O
; using the XL driver with pre-allocated ring buffers.
; XLI0 is the input ring buffer; XLO0 is the output ring buffer.

    MOV      #XL,TTISDB+SD.NAM    ; Put input ring buffer
    MOV      #'I0,TTISDB+SD.NAM+2  ; name into SDB
    MOV      #' ,TTISDB+SD.NAM+4  ; Pad with 2 spaces

    MOV      #XL,TTOSDB+SD.NAM    ; Put output ring buffer
    MOV      #'O0,TTOSDB+SD.NAM+2  ; name into SDB
    MOV      #' ,TTOSDB+SD.NAM+4  ; Pad with 2 spaces

; This section creates the reply queue semaphore

    crst$ #RSDB,styp= ST.QSM,satr= SA$QUO!SA$PRO,value= 0
    BCC     10$                 ; Success

    JSR      PC,$DEGCS          ; Declare exception

; Now setup the SDB for SRTDSP(the directory service process)
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
10$:    MOV      #"$R,DSPSDB+SD.NAM      ; Set queue name into SDB
        MOV      #"$TD,DSPSDB+SD.NAM+2
        MOV      #"$SP,DSPSDB+SD.NAM+4
        .BR     MAIN                  ; Enter main code

        .sbttl * Start of Main Code

; This is the main loop of the program. First, the FIB named
; INFIL is cleared of any previous contents. Then the user is
; prompted for an input file specification. This specification
; is passed the PARSE$ macro for parsing by the SPARSE
; subroutine. If this file specification is legal, the program
; attempts to look up the file by calling SQIO. If the
; specification is not legal, the user is informed of the
; error and then prompted for new specification.

.ENABLE LSB

MAIN:
        MOV      #INFIL,R1           ; First, clear out the FIB
        MOV      #FIBSIZE/2,R0          ; Get word count(bytes/2)
1$:    CLR      (R1)+             ; Clear a word, increment,
        SOB      R0,1$              ; and loop, until done

        .print  #IPROMP            ; Print the first prompt
        .gtlin   #LNUUF             ; Read file name, len in R0
        BCS     DIE                ; Fatal error in terminal I/O

        MOVB    #'D,INFIL+FB.EXT    ; The default extension
        MOV      #AT,INFIL+FB.EXT+1  ; will be .DAT

; Parse the file specification and put the parsed specification in the
; file identification block, beginning with the unit number.

        parse$  buff= LNUUF,len=R0,dest= INFIL+FB.UNI
        BCC     5$                  ; File specification OK

        .print  #EIFS               ; File specification error
        BR      MAIN                ; Go try again

; In this section the input file is opened. First, the driver
; SDB name is taken from the file identification block. A
; dollar sign is prefixed to this name because the semaphore
; name is the device name with the dollar sign. For example,
; the device DYAO: is accessed by sending messages to the SDB named
; $DYA with a @ in FB.UNI of the associated file identification block.

; The GVAL$ primitive request is used to obtain the rest of the structure
; descriptor information for the SDB used to access the device driver
; process. If the device name is illegal, an error is returned, otherwise
; a channel has been opened to the file that is now defined by the file
; identification block.

; This section also prepares for a reply and buffering.
; Finally, a message is sent to RTDSP to establish access to the file.

5$:    CLR      TMPSDB            ; Setup for driver info
        MOV      #TMPSDB+SD.NAM,R1  ; Address of area
        MOVB   #'$, (R1)+           ; Add a $ to the device
        MOVB   INFIL+FB.DVC,(R1)+   ; name in FB.DVC, and
        MOVB   INFIL+FB.DVC+1,(R1)+  ; append the controller
        MOVB   INFIL+FB.CTL,(R1)+   ; name from FB.CTL. Pad
        MOV      #'@,(R1)+           ; with blanks.
```

## MACRO-11 FILE SYSTEM EXAMPLE

; Fill in the index and the sequence number for  
; the SDB name in TMPSDB.

```
gval$ s sdb= TMPSDB,type= INBUF,val= INBUF
BCC      7$                                ; Success
.print  #ENSD
BR      MAIN                               ; No such device error
                                         ; Try again
```

; This section copies the driver SDB information into the FIB  
; then sets up driver and reply SDB, and input buffers  
; then opens input file

```
7$:    MOV      #TMPSDB,R1                ; Put address of SDB in R1
       MOV      #INFIL+FB.DVR,R2            ; and address of field in R2
       MOV      (R1)+,(R2)+                ; Copy word first,
       MOV      (R1)+,(R2)+                ; second,
       MOV      (R1)+,(R2)+                ; and third words
       MOV      #INBUF,INFIL+FB.BFP        ; Set buffer address
       MOV      #IBFSZ,INFIL+FB.BSZ        ; Set buffer size
       MOV      #INBUF+IBFSZ,INFIL+FB.END ; And end of buffer
                                         ; For now, FB.PTR FB.RSZ,
                                         ; FB.RPB, FB.BF2 can be
                                         ; ignored.
```

; Copy the reply SDB information into  
; the reply semaphore field of the FIB

```
MOV      #RSDB,R1                ; Put address of SDB in R1
MOV      #INFIL+FB.REP,R2            ; and address of field in R2
MOV      (R1)+,(R2)+                ; Copy first
MOV      (R1)+,(R2)+                ; second,
MOV      (R1)+,(R2)+                ; and third words.
```

; CLR INFIL+FB.STA ; Set DSP flag word in FIB

; Now, look up the input file.

```
qio$    sdb= DSPSDB,fib= INFIL,func= IF$LOK,seq= 0,reply=YES
BCC      10$                                ; Successful I/O
.print  #EFNF
JMP      MAIN                               ; File can't be opened
                                         ; Start over
DIE:   STPC$S                            ; Terminal I/O failure, so halt
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
; This section clears the output FIB, prompts for an output file name
; and parses the file specification received.

10$:    MOV      #OUTFIL,R1          ; Put FIB address in R1
        MOV      #FIBSZ/2,R0          ; Get word count(bytes/2)
11$:    CLR      (R1)+            ; Clear next word
        SOB      R0,11$             ; loop until done

        .print   #OPROMP           ; Prompt for file specification
        .gtlin   #LNBUF             ; Read it, length is in R0

        BCS      DIE               ; fatal terminal I/O failure

        MOVB    #'D,OUTFIL+FB.EXT  ; The default extension
        MOV     #'AT,OUTFIL+FB.EXT+1 ; will be .DAT

; The parsed file specification is put in the FIB

        parse$  buff= LNBUF,len=R0,dest= OUTFIL+FB.UNI
        BCC    15$                ; Successful parse

        .print   #EIFS              ; Illegal specification error
        BR     10$                ; Go back and try again

; This section creates the specified file based on the file
; specification the the file identification block OUTFIL.

15$:    CLR      TMPSDB            ; Set up driver information
        MOV      #TMPSDB+SD.NAM,R1  ; Address of area
        MOVB   #'$, (R1)+          ; For driver semaphore name
        MOVB   OUTFIL+FB.DVC,(R1)+ ; use device name in FIB
        MOVB   OUTFIL+FB.DVC+1,(R1)+ ; and controller id
        MOVB   OUTFIL+FB.CTL,(R1)+ ; and controller id
        MOV     #", (R1)+

; Fill in index and sequence number for driver SDB

        gval$  sdb= TMPSDB,type= OUTBUF,val= OUTBUF
        BCC    17$                ; Successful ??

        .print   #ENSD              ; No, print error message
        BR     10$                ; and try again

; Copy Driver SDB into FIB driver SDB field

17$:    MOV      #TMPSDB,R1          ; Put SDB address in R1, and
        MOV      #OUTFIL+FB.DVR,R2  ; FIB field address in R2
        MOV      (R1)+,(R2)+          ; Copy first word,
        MOV      (R1)+,(R2)+          ; second word,
        MOV      (R1)+,(R2)+          ; and third word.

        MOV      #OUTBUF,OUTFIL+FB.BFP ; Put buffer address in FIB
        MOV      #OBFSZ,OUTFIL+FB.BSZ  ; Put buffer size in FIB
        MOV      #OUTBUF+OBFSZ,OUTFIL+FB.END ; And end of buffer
                                         ; Ignore FB.PTR, FB.RSZ,
                                         ; FB.RPB, and FB.BF2 for now.
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
; Copy reply SDB into the reply SDB field of the FIB

    MOV      #RSDB,R1           ; Put SDB address into R1,
    MOV      #OUTFIL+FB.REP,R2   ; address of FIB field in R2
    MOV      (R1)+,(R2)+        ; Move first word,
    MOV      (R1)+,(R2)+        ; second word,
    MOV      (R1)+,(R2)+        ; and then third word

; Create(enter) the file

    qio$ sdb= DSPSDB,fib= OUTFIL,func= IF$ENT,seq= 0,reply=YES
    BCC      20$                ; Successful I/O, so skip error

    .print  #ECCO               ; Can't open output file
    JMP      10$                ; Get another specification

; The file is ready to be read, so input can begin.
; First, a buffer (1 block) of input data is read.

20$:   qio$     fib= INFIL,func= IF$RDV ; Read a block
       BCC      25$                ; Success, skip error

       .print  #EERF               ; Error reading file
       BR       40$                ; Done, go close the files

; Now write it to the output file

25$:   BIT      #$EOF,INFIL+FB.STA  ; Test for end of file
       BNE      40$                ; If at EOF, close files

       MOV      #INBUF,R1          ; R1= Input buffer address
       MOV      #OUTBUF,R2          ; R2= Output buffer address
       MOV      #FIBSIZ/2,R0          ; Length in words into R0
26$:   MOV      (R1)+,(R2)+        ; Copy input to output
       SOB      R0,26$              ; until all is copied

       qio$     fib= OUTFIL,func= IF$WTW; Write to output
       BCC      30$                ; Successful I/O

       .print  #EEWF               ; Error writing file
       BR       40$                ; Done

30$:   BIT      #$EOF,OUTFIL+FB.STA  ; Test EOF bit.
       BNE      35$                ; If set, branch to error

       INC      INFIL+FB.BNO        ; Next block of input
       INC      OUTFIL+FB.BNO        ; and of output
       BR       20$                ; Go get another block

; End of output file was passed, so print error message.

35$:   .print  #EWPE               ; Print past EOF message

; Close input and output files

40$:   qio$     sdb= DSPSDB,fib= INFIL, func= IF$CLS
       qio$     sdb= DSPSDB,fib= OUTFIL,func= IF$CLS

; or    JMP      MAIN                ; Prompt for more files
       stpc$$s  pdb= 0                ; could just exit.

       .DSABL  LSB
       .sbttl  Subroutines for I/O
       .sbttl  * $STOUT - Types message to TTOSDB Ring buffer
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
;+
; $TTOUT - prints messages at the console.
;
; Input:
;           R0 -> .ASCII message
;
; This subroutine finds the length of a message,
; then uses the PELM$ primitive to write it to the console.
; This routine assumes the use of an XL driver using
; the pre-allocated buffers.
;
; If the carry bit is set by Pelm$, this routine returns
; with the carry bit still set, thus indicating a fatal terminal
; I/O error.
;-

$TTOUT:
    push$   R0          ; Save the pointer on the stack
1$:    TSTB   (R0)+      ; Increment R0
    BNE    1$          ; until null (0) is found
    SUB    @SP,R0       ; Get char count

; Put the character to the terminal

    pelm$ sdb= TTOSDB,bufptr=2(SP),bufcnt=R0

    pop$   R0          ; Restore R0
    RETURN           ; Done, carry set on error

.sbttl * $TTIN - Reads a line from TTISDB Ring Buffer

;+
; $TTIN  Reads input line
;
; Input:
;           R0 -> Line buffer
; Output:=
;           R0 = Length of input line
;
; This routine reads characters from the ring buffer until <CR>
; is encountered. Characters are echoed to TTOSDB. A <LF> is
; sent after <CR>.

; Gelm$S and Pelm$S set the carry bit set if an I/O failure
; occurs. If the carry bit returns from those primitives set,
; this routine returns immediately with the carry still set.
;-

$TTIN:
    push$   R1          ; Save the register on the stack
    push$   R0          ; Save the buffer pointer
    MOV    R0,R1         ; Copy the buffer pointer

; Get an element from the terminal by using Get Element (Gelm$S) to
; get a character from the input ring buffer of the terminal's serial
; line.

; Get one character
```

## MACRO-11 FILE SYSTEM EXAMPLE

```
1$:    gelm$S  sdb= TTISDB,bufptr=R1,bufcnt= 1
      BCS     20$          ; Return error

      MOVB    (R1),R0        ; Get character into R0
      BIC     #^C177,R0       ; Clean up character

      CMPB    #DEL,R0        ; Rubout key?
      BNE     5$          ; Not rubout, so skip

; Rubout last character, if any

      CMP     R1,@SP        ; Any chars to rub?
      BEQ     1$          ; No, just get next one

      DEC     R1          ; Remove last char

; Put the rubout sequence to the terminal by doing put elements to the
; output ring buffer of that serial line.

      char   pelm$S  sdb= TTOSDB,bufptr= BSCH,bufcnt= LBSCH ; Back up a
      BCS     20$          ; Return error

      BR     1$          ; Get another character

; A character was entered, so echo it to the terminal.

5$:    pelm$S  sdb= TTOSDB,bufptr=R1,bufcnt= 1
      BCS     20$          ; Return an error

      BICB    #200,@R1        ; Clean up the character
      CMPB    #CR,@R1        ; End of line?
      BEQ     10$          ; Yes, go terminate line.
      INC     R1          ; Next buffer slot
      BR     1$          ; Get another character

; End of line reached (<CR>), so echo additional <LF> and return

10$:   CLRB    @R1          ; Terminate line with null char

; Put element to output ring buffer

      pelm$S  sdb= TTOSDB,bufptr= LFCH,bufcnt= 1
      BCS     20$          ; Return error
      MOV     R1,R0          ; Now find length of input line
      SUB     (SP)+,R0        ; and put that value in R0
      pop$    R1          ; Restore R1 from stack
      RETURN           ; Subroutine finished

20$:   pop$    R0          ; Restore R0 from stack
      pop$    R1          ; Restore R1 from stack
      RETURN           ; Carry still set implies error
      .end
```

## APPENDIX G

### LSI-11 ANALOG SYSTEM INTERFACE

This appendix describes the MicroPower/Pascal interface to LSI-11 analog I/O systems.

MicroPower/Pascal software supports the use of four analog boards that are I/O options for the LSI-11 family of processors. These boards are the following:

- ADV11-C analog input board
- AAV11-C analog output board
- AXV11-C analog input/output board
- KWV11-C programmable real-time clock

Each board may be used by itself on the LSI-11 bus, but typically each is used with one or more of the other boards to create an analog I/O system.

The ADV11-C analog input board and the AXV11-C analog input/output board convert analog signals to digital data. The ADV11-C and the AXV11-C each accept 16 single-ended analog inputs or 8 differential analog inputs. Each has a programmable gain of 1, 2, 4, or 8 times the input signal. The user connects analog signals to the board through an I/O connector. After the analog-to-digital (A/D) conversion is complete, the user gets the results via a programmed I/O transfer or a serviced interrupt request.

In addition to the input channels, the AXV11-C has two digital-to-analog converters (DACs). Each DAC can be loaded with digital data from the LSI-11 bus to be changed to an analog voltage. The user can specify the format of the input data, as well as the range and polarity of the output voltage.

The AAV11-C analog output board is used exclusively for digital-to-analog (D/A) conversions. The AAV11-C has four DACs, each of which has a separate register that can be written or read in byte or word format, allowing complete use of the LSI-11 instruction set. Each DAC is loaded from the LSI-11 bus to create an analog output voltage at its I/O connector. One of the registers can also be used as a four-bit digital output for control signals, such as CRT intensity, blank, or erase.

The KWV11-C programmable real-time clock has a crystal oscillator and two Schmitt triggers that are often used with the A/D input boards to start A/D conversions. 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.

## LSI-11 ANALOG SYSTEM INTERFACE

The MicroPower/Pascal interface to LSI-11 analog I/O systems consists of the following:

- ADV11-C/AXV11-C (AA) analog input converter handler
- KWV11-C (KW) real-time clock handler
- Real-time clock procedures `Read_counts_wait`, `Read_counts_signal`, `Start_rtclock`, and `Stop_rtclock`, which use the KW handler
- Analog-to-digital (A/D) conversion procedure `Read_analog_wait`, which uses programmed I/O
- Analog-to-digital (A/D) conversion procedures `Read_analog_signal` and `Read_analog_continuous`, which use the AA handler and require the KW handler
- Digital-to-analog (D/A) conversion procedure `Write_analog_wait`, which uses programmed I/O

The sections that follow describe the components of the MicroPower/Pascal interface to LSI-11 analog I/O systems. Section G.1 describes the interface to the KWV11-C real-time clock; Section G.2, the interface to the ADV11-C and AXV11-C analog input converters; and Section G.3, the interface to the AAV11-C and AXV11-C digital-to-analog converters (DACs). Section G.4 provides examples showing how the MicroPower/Pascal interface to LSI-11 analog I/O systems may be used.

### NOTE

The following files are referred to throughout this appendix:

RHSLIB.OBJ	Analog I/O and real-time clock I/O library
RHSDSC.PAS	Analog I/O and real-time clock I/O include file
IOPKTS.PAS	Pascal device I/O include file
KWVINC.PAS	Real-time clock I/O include file

The RHSLIB object library contains the object modules for the real-time clock, A/D conversion, and D/A conversion procedures listed above. The RHSDSC include file externally declares the above-listed procedures and defines data structures used with them. The IOPKTS include file defines device characteristics parameters, error and severity codes, function codes, and request and reply packets for Pascal device I/O requests. The KWVINC include file defines data structures for real-time clock I/O requests.

### G.1 KWVII-C REAL-TIME CLOCK INTERFACE

The MicroPower/Pascal interface to the KWVII-C programmable real-time clock consists of the following:

- KW real-time clock handler
- `Read_counts_wait` procedure, which uses the KW handler
- `Read_counts_signal` procedure, which uses the KW handler
- `Start_rtclock` procedure, which uses the KW handler
- `Stop_rtclock` procedure, which uses the KW handler

The KW handler provides a standard device handler interface to the KWVII-C. See the KW handler section of Chapter 4 for a detailed description of this interface.

The `Read_counts_wait` and `Read_counts_signal` procedures, described in Sections G.1.1 and G.1.2, support the external event modes of the KWVII-C. These procedures 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 `Start_rtclock` procedure, described in Section G.1.3, can be used to set up the real-time clock for interval timing or as a free-running clock used to initiate A/D conversions.

The `Stop_rtclock` procedure, described in Section G.1.4, disables interrupts on the real-time clock.

#### • Read\_counts\_wait

The `Read_counts_wait` procedure provides a synchronous interface to read a "block" of counts via the programmable real-time clock. `Read_counts_wait` starts the I/O request and waits until it is completed to return. The `Read_counts_wait` procedure can be used in conjunction with the KWVII-C programmable real-time clock and its driver 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, which reside on the KWVII-C board. A Schmitt trigger is a voltage threshold detector. When the voltage of an input signal to a Schmitt trigger crosses a specified threshold, the Schmitt trigger generates a pulse. The threshold is set by adjusting a potentiometer. A Schmitt trigger can be set to fire when the threshold is crossed in either a positive or a negative direction, but not both at the same time. The user selects the desired direction of the slope (plus or minus) of the firing signal by flipping a switch.

The KWVII-C has two Schmitt triggers. The primary Schmitt trigger is the second one, ST2. It can be used to cause interrupts, or it can be used to start the clock. The first Schmitt trigger, ST1, can be used only to increment the clock.

## LSI-11 ANALOG SYSTEM INTERFACE

To record the time of external events or the time between external events, a rate is specified for the source: kwy\_1MHz through kwy\_100Hz; or kwy\_line. The clock is started by either the handler setting the GO bit in the CSR or 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. An overrun condition 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. An overflow condition may indicate that too high a rate was specified.

To measure the relative frequency of one event to another, the user specifies kwy\_ST1 as the source. This is the only difference from the above. 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 `Read_counts_wait` procedure is:

```
Read_counts_wait
  ( buffer := array-name;
    number := integer;
    source := kwy-rate;
    base   := base-type;
    start  := start-type;
    state  := status-word );
```

Parameter	Description
array-name	A variable of type ARRAY [first..last : INTEGER] OF INTEGER, indicating the array of integers that the counter is to be copied to after each interrupt.
integer	An integer constant or variable indicating the number of elements to be copied to the array.
kwy-rate	A value indicating the source of the counts. The value kwy_stop is illegal; kwy_1MHz, kwy_100kHz, kwy_10kHz, kwy_1kHz, and kwy_100Hz specify clock ticks at the respective rates; kwy_ST1 specifies counts of events logged on Schmitt trigger 1; and kwy_line specifies clock ticks at the line frequency (50 or 60 Hz). The values for this parameter are defined in the KWVINC include file.
base-type	A value indicating 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. The values for this parameter are defined in the RHSDSC include file.

## LSI-11 ANALOG SYSTEM INTERFACE

start-type	A value indicating 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. The values for this parameter are defined in the RHSDSC include file.
status-word	A variable of type IO\$STATUS indicating where the status of the procedure call is to be returned. The following values can be returned in the high-order 13 bits of the status word: IE\$NORMAL (successful operation), IE\$FUNINV (invalid I/O function code), IE\$DEVALLOC (device allocated), IE\$OVERFLOW (overflow occurred), or IE\$OVERRUN (overrun occurred). The IO\$STATUS data type is defined in the IOPKTS include file.

### G.1.2 Read\_counts\_signal

The Read\_counts\_signal procedure provides an asynchronous interface to read a block of counts via the programmable real-time clock. Read\_counts\_signal starts the I/O request and returns immediately. When the I/O request is completed, the specified semaphore is signaled. The Read\_counts signal procedure can be used in conjunction with the KWV11-C programmable real-time clock and its driver 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, which reside on the KWV11-C board. A Schmitt trigger is a voltage threshold detector. When the voltage of an input signal to a Schmitt trigger crosses a specified threshold, the Schmitt trigger generates a pulse. The threshold is set by adjusting a potentiometer. A Schmitt trigger can be set to fire when the threshold is crossed in either a positive or a negative direction, but not both at the same time. The user selects the desired direction of the slope (plus or minus) of the firing signal by flipping a switch.

The KWV11-C has two Schmitt triggers. The primary Schmitt trigger is the second one, ST2. It can be used to cause interrupts, or it can be used 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 rate is specified for the source: kwt\_1MHz through kwt\_100Hz; or kwt\_line. The clock is started by either the handler setting the GO bit in the CSR or 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. An overrun condition 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. An overflow condition may indicate that too high a rate was specified.

## LSI-11 ANALOG SYSTEM INTERFACE

To measure the relative frequency of one event to another, the user specifies `kvv_ST1` as the source. This is the only difference from the above. 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 `Read_counts_signal` procedure is:

```
Read_counts_signal
  ( buffer := array-name;
    number := integer;
    source := kvv-rate;
    base   := base-type;
    start  := start-type;
    reply  := structure-id );
```

Parameter	Description
array-name	A variable of type <code>ARRAY [first..last : INTEGER] OF INTEGER</code> , indicating the array of integers that the counter is to be copied to after each interrupt.
integer	An integer constant or variable indicating the number of elements to be copied to the array.
<code>kvv-rate</code>	A value indicating the source of the counts. The value <code>kvv_stop</code> is illegal; <code>kvv_1MHz</code> , <code>kvv_100kHz</code> , <code>kvv_10kHz</code> , <code>kvv_1kHz</code> , and <code>kvv_100Hz</code> specify clock ticks at the respective rates; <code>kvv_ST1</code> specifies counts of events logged on Schmitt trigger 1; and <code>kvv_line</code> specifies clock ticks at the line frequency (50 or 60 Hz). The values for this parameter are defined in the <code>KWVINC</code> include file.
<code>base-type</code>	A value indicating the base for counting. The value <code>rtc_continuous</code> specifies that the count at any event is continuous from the first event; <code>rtc_zero_base</code> specifies that the count resets to 0 after each event. The values for this parameter are defined in the <code>RHSDSC</code> include file.
<code>start-type</code>	A value indicating how the clock is to be started. The value <code>immediate</code> specifies that the clock is to be started immediately; <code>event</code> specifies that the clock is to be triggered by an external event on Schmitt trigger 2. The values for this parameter are defined in the <code>RHSDSC</code> include file.
<code>structure-id</code>	A constant or variable of predefined type <code>STRUCTURE_ID</code> , indicating the queue semaphore that the <code>final_status</code> message is to be sent to when the I/O request is completed. The message is a reply packet from the real-time clock handler of type <code>IOSREPLY_PKT</code> . Included in the packet is a 16-bit status word of type <code>IO\$STATUS</code> . The following values can be returned in the high-order 13 bits of the status word: <code>IE\$NORMAL</code> (successful operation), <code>IE\$UNINV</code> (invalid I/O function code), <code>IE\$DEVALLOC</code> (device allocated), <code>IE\$STOP</code> (processing stopped by request), or <code>IE\$OVERRUN</code> (overrun occurred). The <code>IOSREPLY_PKT</code> and <code>IO\$STATUS</code> data types are defined in the <code>IOPKTS</code> include file.

## G.1.3 Start\_rtclock

The Start\_rtclock procedure is used in conjunction with the KWW11-C programmable real-time clock and its driver to set 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. If signaling is specified, the semaphore designated by the user will be signaled at the end of the specified interval or at the end of each such interval if repeated intervals are specified. If no signaling is specified, clock interrupts are disabled, and the clock runs freely. This free-running mode can be used to trigger A/D conversions if desired.

The syntax for calling the Start\_rtclock procedure is:

```
Start_rtclock
  ( source := kww-rate;
    counts := integer;
    single := boolean-1;
    start := start-type;
    signals := boolean-2;
    timer := structure-id;
    state := status-word );
```

Parameter	Description
kww-rate	A value indicating the source of the counts. The value kww_stop is illegal; kww_1MHz, kww_100kHz, kww_10kHz, kww_1kHz, and kww_100Hz specify clock ticks at the respective rates; kww_ST1 specifies counts of events logged on Schmitt trigger 1; and kww_line specifies clock ticks at the line frequency (50 or 60 Hz). The values for this parameter are defined in the KWWINC include file.
integer	An integer constant or variable indicating the number of clock ticks at the specified rate (kww-rate) until the counter overflows.
boolean-1	A Boolean constant or variable indicating whether the clock is to run for a single interval (TRUE) or continuously at the specified rate (FALSE).
start-type	A value indicating 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. The values for this parameter are defined in the RHSDSC include file.
boolean-2	A Boolean constant or variable 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.
structure-id	A constant or variable of predefined type STRUCTURE_ID, indicating the semaphore (binary or counting) to be signaled at the end of each interval.

## LSI-11 ANALOG SYSTEM INTERFACE

**status-word** A variable of type IO\$STATUS indicating where the status of the procedure call is to be returned. The following values can be returned in the high-order 13 bits of the status word: IE\$NORMAL (successful operation), IE\$UNINV (invalid I/O function code), IE\$DEVALLOC (device allocated), IE\$STOP (processing stopped by request), IE\$OVERFLOW (overflow occurred), or IE\$OVERRUN (overrun occurred). Overrun errors subsequent to the first will not be indicated. The IO\$STATUS data type is defined in the IOPKTS include file.

### G.1.4 Stop\_rtclock

The **Stop\_rtclock** procedure stops the KWV11-C programmable real-time clock by disabling interrupts on the device.

The syntax for calling the **Stop\_rtclock** procedure is:

```
Stop_rtclock;
```

This procedure has no arguments. If a **Read\_counts** signal request is in progress when this procedure is invoked, the "final status" message will be sent to the "signal" queue semaphore to indicate that the clock was stopped.

## G.2 ANALOG-TO-DIGITAL (A/D) CONVERSION INTERFACE

The MicroPower/Pascal interface to the ADV11-C and AXV11-C analog input converters consists of the following:

- AA analog input converter handler
- **Read\_analog\_wait** procedure, which uses programmed I/O
- **Read\_analog\_signal** procedure, which uses the AA handler and requires the KW handler
- **Read\_analog\_continuous** procedure, which uses the AA handler and requires the KW handler

The AA handler provides a standard device handler interface to the ADV11-C and the AXV11-C. See Section 4.2 for a detailed description of this interface.

The **Read\_analog\_wait** procedure, described in Section G.2.1, supports synchronous, block-mode A/D conversion. This procedure does not use a device handler.

The **Read\_analog\_signal** procedure, described in Section G.2.2, supports asynchronous, block-mode A/D conversion. This procedure uses the real-time clock to initiate A/D conversions.

The **Read\_analog\_continuous** procedure, described in Section G.2.3, supports continuous A/D conversion. Like **Read\_analog\_signal**, this procedure uses the real-time clock to initiate A/D conversions.

## LSI-11 ANALOG SYSTEM INTERFACE

### G.2.1 Read\_analog\_wait

The `Read_analog_wait` procedure provides a synchronous interface to read a block of values via the analog-to-digital converters. `Read_analog_wait` uses programmed I/O to read 1 to 16 channels of analog data with the specified gain.

The syntax for calling the `Read_analog_wait` procedure is:

```
Read_analog_wait
  ( buffer      := array-name;
    chan_ptr    := ad-control_ptr;
    state       := status-word );
```

#### Parameter Description

`array-name` A variable of type `ARRAY [first..last : INTEGER] OF INTEGER` indicating the array of integers that the converted data is to be copied to after each interrupt.

`ad-control_ptr` A pointer to a control structure (see the note below) that specifies the number of channels to sample and the channel number and gain select for each channel to be sampled.

`status-word` A variable of type `IOSSTATUS` indicating where the status of the request is to be returned. The following values can be returned in the high-order 13 bits of the status word: `IENORMAL` (successful operation) and `IEOVERRUN` (overrun occurred). The `IOSSTATUS` data type is defined in the `IOPKTS` include file.

#### NOTE

The following data types, which are defined in the `RHSDSC` include file, are used in connection with the `ad-control_ptr` parameter to the `Read_analog_wait` procedure:

```
TYPE
  mpx_addr = #..15; {Channel numbers}

  ad_gain = (
    ad_gain_1,
    ad_gain_2,
    ad_gain_4,
    ad_gain_8 );

  ad_chan_desc = PACKED RECORD
    chan_num      : #BIT(4) mpx_addr;
    filler        : #BIT(8) 0..255; {Reserved}
    gain_sel     : #BIT(4) ad_gain;
  END;

{** User must point to a record of this type **}
  ad_control_type = PACKED RECORD
    num_chan      : INTEGER; {Number of channels}
    chan_ctrl    : ARRAY [1..16] OF ad_chan_desc;
  END;

  ad_control_ptr = ^ ad_control_type;
```

## LSI-11 ANALOG SYSTEM INTERFACE

### G.2.2 Read\_analog\_signal

The `Read_analog_signal` procedure provides an asynchronous interface to read a block of values via the analog-to-digital converters. `Read_analog_signal` uses the analog-to-digital converter (AA) handler and requires the KWV11-C programmable real-time clock -- or a user-supplied alternative to the KWV11-C -- along with its device handler.

The `Read_analog_signal` procedure initiates the analog conversion requested by the user and returns immediately. This procedure requires that the A/D conversion on the first channel for each sample be initiated by an external mechanism. This external mechanism could be a Schmitt trigger on the KWV11-C or the clock signal on the KWV11-C. Since the analog-to-digital converters and the real-time clock are on separate modules, the user must make an electrical connection between the two so that the A/D module can sense the real-time clock's signal. For instance, if the real-time clock's output signal (CLK\_OVL -- clock overflow) is used to initiate the conversion on the first specified channel, the signal (CLK\_OVL) generated on the KWV11-C board must be wired directly to the input line (RTC IN L -- real-time clock input, asserted low) on the A/D converter board.

When the conversion of the first specified channel occurs, the A/D converter requests an interrupt. The ISR (interrupt service routine) reads the first value and copies it to the user-specified buffer. If additional channels are to be read, they are read synchronously under program control and are copied directly to the user buffer.

One to sixteen channels can be sampled after each interrupt. Overlapping calls to `Read_analog_signal` are queued to allow multibuffer operations.

The syntax for calling the `Read_analog_signal` procedure is:

```
Read_analog_signal
  ( buffer   := array-name;
    number   := integer;
    chan_ptr := ad-control-ptr;
    trigger   := trigger-type;
    reply    := structure-id );
```

Parameter	Description
array-name	A variable of type ARRAY [first..last : INTEGER] OF INTEGER, indicating the array of integers that the converted data is to be copied to after each interrupt.
integer	An integer constant or variable indicating the number of sample points (1 to 16 conversions per sample point).

## LSI-11 ANALOG SYSTEM INTERFACE

**ad-control-ptr** A pointer to a control structure (see the note below) that specifies the number of channels to sample and the channel number and gain select for each channel to be sampled.

**trigger-type** A value indicating how the A/D conversion is to be initiated. The value external\_event specifies that the conversion is to be initiated by an external event; real\_time\_clock specifies that the conversion is to be initiated by a real-time clock. The values for this parameter are defined in the RHSDSC include file.

**structure-id** A constant or variable of predefined type STRUCTURE\_ID, indicating the queue semaphore that the final\_status message is to be sent to when the I/O request is completed. The message is a reply packet from the A/D handler of type IO\$REPLY\_PKT. Included in the packet is a 16-bit status word of type IO\$STATUS. The following values can be returned in the high-order 13 bits of the status word: IE\$NORMAL (successful operation), IE\$STOP (processing stopped by request), or IE\$OVERRUN (overrun occurred). The IO\$REPLY\_PKT and IO\$STATUS data types are defined in the IOPKTS include file.

### NOTE

The following data types, which are defined in the RHSDSC include file, are used in connection with the ad-control-ptr parameter to the Read\_analog\_signal procedure:

```
TYPE
  mpx_addr = 0..15; {Channel numbers}

  ad_gain = (
    ad_gain_1,
    ad_gain_2,
    ad_gain_4,
    ad_gain_8 );

  ad_chan_desc = PACKED RECORD
    chan_num      : [BIT(4)] mpx_addr;
    filler        : [BIT(8)] 0..255; {Reserved}
    gain_sel      : [BIT(4)] ad_gain;
  END;

{** User must point to a record of this type **}
  ad_control_type = PACKED RECORD
    num_chan      : INTEGER; {Number of channels}
    chan_ctrl    : ARRAY [1..16] OF ad_chan_desc;
  END;

  ad_control_ptr = ^ ad_control_type;
```

## LSI-11 ANALOG SYSTEM INTERFACE

### G.2.3 Read\_analog\_continuous

The Read\_analog\_continuous procedure provides an asynchronous interface to continuously read a block of values via the analog-to-digital converters and to put them in a user-created ring buffer. Read\_analog\_continuous uses the analog-to-digital converter (AA) handler and requires the KVV11-C programmable real-time clock -- or a user-supplied alternative to the KVV11-C -- along with its device handler.

The Read\_analog\_continuous procedure starts another process to perform the analog conversion requested by the user and returns immediately. This procedure requires that the A/D conversion on the first channel for each sample be initiated by an external mechanism. This external mechanism could be a Schmitt trigger on the KVV11-C or the clock signal on the KVV11-C. Since the analog-to-digital converters and the real-time clock are on separate modules, the user must make an electrical connection between the two so that the A/D module can sense the real-time clock's signal. For instance, if the real-time clock's output signal (CLK\_OVL -- clock overflow) is used to initiate the conversion on the first specified channel, the signal (CLK\_OVL) generated on the KVV11-C board must be wired directly to the input line (RTC IN L -- real-time clock input, asserted low) on the A/D converter board.

When the conversion of the first specified channel occurs, the A/D converter requests an interrupt. The ISR (interrupt service routine) reads the first value and copies it to a local buffer. If additional channels are to be read, they are read synchronously under program control and are copied to the same local buffer. After the specified channels have been read, the entire local buffer -- consisting of a status word, as defined in the IOPKTS include file, followed by an array of converted values -- is put into the specified ring buffer. (Returning the requested values all at once minimizes data point skew and increases performance.)

One to sixteen channels can be sampled after each interrupt. The ring buffer, which must be created prior to making a request, must be large enough to contain two complete samples -- that is, in bytes, four times the number of channels to be sampled, plus four.

Overlapping calls to Read\_analog\_continuous will cause the previous request to be terminated.

The syntax for calling the Read\_analog\_continuous procedure is:

```
Read_analog_continuous
  ( ring      := structure-id;
    chan_ptr  := ad-control-ptr;
    trigger   := trigger-type;
    state     := status-word );
```

Parameter	Description
structure-id	A constant or a variable of predefined type STRUCTURE_ID, indicating the ring buffer in which the sample points are to be placed after each interrupt.
ad-control-ptr	A pointer to a control structure (see the note below) that specifies the number of channels to sample and the channel number and gain select for each channel to be sampled.

## LSI-11 ANALOG SYSTEM INTERFACE

**trigger-type** A value indicating how the A/D conversion is to be initiated. The value external\_event specifies that the conversion is to be initiated by an external event; real\_time\_clock specifies that the conversion is to be initiated by a real-time clock. The values for this parameter are defined in the RHSDSC include file.

**status-word** A variable of type IOSSTATUS indicating where the status of the procedure call is to be returned. The following values can be returned in the high-order 13 bits of the status word: IESNORMAL (successful operation) and IESINVPARAM (invalid parameter). The IOSSTATUS data type is defined in the IOPKTS include file.

### NOTE

The following data types, which are defined in the RHSDSC include file, are used in connection with the ad-control\_ptr parameter to the Read\_analog\_continuous procedure:

```
TYPE
  mpx_addr = 0..15; {Channel numbers}
  ad_gain = (          {A/D gain values}
    ad_gain_1,
    ad_gain_2,
    ad_gain_4,
    ad_gain_8 );
  ad_chan_desc = PACKED RECORD
    chan_num      : [BIT(4)] mpx_addr;
    filler        : [BIT(8)] 0..255; {Reserved}
    gain_sel     : [BIT(4)] ad_gain;
  END;
{** User must point to a record of this type **}
  ad_control_type = PACKED RECORD
    num_chan     : INTEGER; {Number of channels}
    chan_ctrl   : ARRAY [1..16] OF ad_chan_desc;
  END;
  ad_control_ptr = ^ ad_control_type;
```

### G.3 DIGITAL-TO-ANALOG (D/A) CONVERSION INTERFACE: WRITE\_ANALOG\_WAIT

The MicroPower/Pascal interface to the AAV11-C and AXV11-C digital-to-analog converters (DACs) consists of the MicroPower/Pascal procedure Write\_analog\_wait, which supports D/A conversion via programmed I/O transfer. This procedure does not use a device handler. (Note, however, that for the sake of consistency with the real-time clock and A/D conversion interfaces, a handler-style prefix file, ABPEX.PAS, is provided for the D/A conversion interface. For a detailed description of ABPEX.PAS, see the MicroPower/Pascal-RT System User's Guide or the MicroPower/Pascal-RSX/VMS System User's Guide.)

## LSI-11 ANALOG SYSTEM INTERFACE

The Write\_analog\_wait procedure interfaces with the digital-to-analog converters on the AAV11-C and AXV11-C analog I/O boards. Write\_analog\_wait uses programmed I/O to write one to four (AAV11-C) or one to two (AXV11-C) values from a buffer to one or more D/A channels.

The syntax for calling the Write\_analog\_wait procedure is:

```
Write_analog_wait
  ( channels  := channel-array;
    buffer    := array-name;
    state     := status-word );
```

Parameter	Description
channel-array	A variable of type ARRAY [chanl..num_chan : four] OF INTEGER (four = 0..3), indicating the array of integers that specify which channel the corresponding value in the second user-specified buffer is to be written to; for example, buffer[2] is written to channels[2].
array-name	A variable of type ARRAY [vall..num_values : four] OF INTEGER (four = 0..3), indicating the array of integers that are to be written to the D/A converters as specified in the channels array.
status-word	A variable of type IOSSTATUS indicating where the status of the request is to be returned. The following values can be returned in the high-order 13 bits of the status word: IESNORMAL (successful operation) and IESINVPARAM (invalid parameter). The IOSSTATUS data type is defined in the IOPKTS include file.

### G.4 EXAMPLES

The following examples illustrate how the procedures described in the preceding sections might be used. Section G.4.1 illustrates timed A/D conversion when the KWV11-C is not available; Section G.4.2, timed D/A conversion when the KWV11-C is not available; Section G.4.3, continuous A/D conversion; and Section G.4.4, asynchronous, block-mode A/D conversion.

#### NOTE

The generic device specification "mpp-lib:", which appears in Pascal %INCLUDE statements throughout this section, indicates that you should specify the logical device/directory on which user-relevant MicroPower/Pascal files reside in your host system. The installation defaults for each type of host system are as follows:

## LSI-11 ANALOG SYSTEM INTERFACE

Host	Installation Default
RT-11 (hard disk)	LB:
RT-11 (diskette)	None (you must supply device and directory)
RSX-11M/M-Plus	MP:[2,10]
VAX/VMS	MICROPOWER\$LIB:

### G.4.1 Timed A/D Conversion When the KWV11-C Is Not Available

```
%INCLUDE 'mpp-lib:IOPKTS'  
%INCLUDE 'mpp-lib:CLKLIB'  
%INCLUDE 'mpp-lib:RHSDSC'  
%INCLUDE 'mpp-lib:KWVINC'  
TYPE  
    sample = ARRAY [1..4] of INTEGER;      { A single sample of 4  
                                              channels of data }
```

## LSI-11 ANALOG SYSTEM INTERFACE

```

VAR
    clock_message : msg_sig_sem;
    signal_semaphore : structure_desc;
    init : BOOLEAN;
    data : ARRAY [1..128] of sample;
    index : INTEGER;
    chan_data : ad_control_type
    status : IOSSTATUS

BEGIN
    initclock;                                { Standard clock library
                                                initialization: creates
                                                the necessary semaphores
                                                and sets the current date
                                                and time as supplied by
                                                the operator }

    init := CREATE_BINARY_SEMAPHORE
        ( desc := signal_semaphore );          { Create the semaphore that
                                                is to be signaled
                                                periodically by the clock
                                                process }

    WITH chan_data DO
        BEGIN
            num_chan := 4;                      { Number of channels to
                                                sample at each time
                                                interval }

            chan_ctrl[1].chan_num := 2;           { First channel sampled is
                                                channel 2 }

            chan_ctrl[1].gain_sel := ad_gain_1;   { Set gain }
            chan_ctrl[2].chan_num := 10;           { Second channel ... }
            chan_ctrl[2].gain_sel := ad_gain_8;   { Set gain }
            chan_ctrl[3].chan_num := 13;           { Third channel ... }
            chan_ctrl[3].gain_sel := ad_gain_1;   { Set gain }
            chan_ctrl[4].chan_num := 15;           { Fourth channel ... }
            chan_ctrl[4].gain_sel := ad_gain_2;   { Set gain }

        END;
        signal_periodic                         { Request clock process to }
        ( clock_message,                        { signal binary semaphore }
          signal_semaphore,                   { every 10 clock ticks }

FOR index := 1 to 128 DO
BEGIN
    WAIT (desc := signal_semaphore);         { Wait for time interval to
                                                expire }

    Read_analog_wait
        ( data[index],
          ADDRESS(chan_data),
          status );
    IF status.error_code <> IESNORMAL
        THEN punt;                           { Take a sample }
                                            { of four channels }
                                            { as described here }
                                            { with status returned here }

    END;
    :                                     { Do something with the buffer of data }
    :

```

{ Message for clock process }

{ Semaphore to be signaled by the clock process }

{ Indicates initialization success or failure }

{ Buffer for data }

{ FOR loop index }

{ Control structure specifying the number of channels and which channels to sample }

{ Status of conversion upon return }

## G.4.2 Timed D/A Conversion When the KWV11-C Is Not Available

```

%INCLUDE 'mpp-lib:TOPRTS'
%INCLUDE 'mpp-lib:CLKLIB'
%INCLUDE 'mpp-lib:RHSDSC'
%INCLUDE 'mpp-lib:KWVINC'

TYPE
    chan_dsc = ARRAY [0..3] of INTEGER;
    sample = ARRAY [0..3] of INTEGER;

VAR
    clock_message : msg_sig_sem;
    signal_semaphore : structure_desc;
    init : BOOLEAN;
    index : INTEGER;
    status : IOSSTATUS;
    d_a_control : chan_dsc;
    data : ARRAY [1..128] of sample;
BEGIN
    initclock;
    init := CREATE_BINARY_SEMAPHORE
        ( desc := signal_semaphore );
    FOR index := 0 to 3 DO
        d_a_control[index] := index;
    signal_periodic
        ( clock_message,
          signal_semaphore,
          10 );
    ;
    ;
    ; Data is collected/generated and stored in the array data
    ;
    FOR index := 1 to 128 DO
    BEGIN
        WAIT (desc := signal_semaphore);
        Write_analog_wait
            ( d_a_control,
              data[index],
              status );
        IF status.error_code <> IFSNORMAL
            THEN punt;
    END;
    ;
    ;
    ;

```

( List of 4 channels to output data )  
 ( A single sample of 4 channels of data )  
 ( Message for clock process )  
 ( Semaphore to be signalled by the clock process )  
 ( Indicates initialization success or failure )  
 ( FOR loop index )  
 ( Status of call upon return )  
 ( Array of output channel descriptors )  
 ( Buffer for data )  
 ( Standard clock library initialization: creates the necessary semaphores and sets the current date and time as supplied by the operator )  
 ( Create the semaphore that is to be signalled periodically by the clock process )  
 ( Build D/A control structure )  
 ( Request clock process to signal binary semaphore )  
 ( every 10 clock ticks )  
 ( Wait for time interval to expire )  
 ( Write four D/A channel's )  
 ( as described here )  
 ( with this data )  
 ( with status returned here )  
 ( call procedure on error )

## G.4.3 Continuous A/D Conversion

```

%INCLUDE 'mpp-lib:IOPKTS'
%INCLUDE 'mpp-lib:RHSDSC'
%INCLUDE 'mpp-lib:KVVINC'
TYPE
  sample = RECORD
    status : IOSSTATUS;
    .
    data : ARRAY [1..4] of INTEGER;
  END;
VAR
  ok : BOOLEAN;
  analog_data : sample;
  index : INTEGER;
  chan_data : ad_control_type;

status : IOSSTATUS;
ring_buf : RING_BUFFER_DESC;
clk_status : IOSSTATUS;
dummy : STRUCTURE_ID;
BEGIN
  ok := CREATE_RING_BUFFER (
    input_order := FIFO,
    input_mode := RECORD_MODE,
    output_order := FIFO,
    output_mode := RECORD_MODE,
    size := 40,
    desc := ring_buf );
  IF NOT ok THEN punt;
  WITH chan_data DO
    BEGIN
      num_chan := 4;
      chan_ctrl[1].chan_num := 2;           { Number of channels to
                                             sample at each time
                                             interval }
      chan_ctrl[1].gain_sel := ad_gain_1;   { First channel sampled is
                                             channel 2 }
      chan_ctrl[2].chan_num := 10;          { Set gain }
      chan_ctrl[2].gain_sel := ad_gain_1;   { Second channel ... }
      chan_ctrl[3].chan_num := 13;          { Set gain }
      chan_ctrl[3].gain_sel := ad_gain_1;   { Third channel ... }
      chan_ctrl[4].chan_num := 15;          { Set gain }
      chan_ctrl[4].gain_sel := ad_gain_1;   { Fourth channel ... }
    END;
  Read_analog_continuous
    ( ring_buf,
      ADDRESS(chan_data),
      real_time_clock,
      status );
  IF status.error_code <> IESNORMAL
    THEN punt;
  { Connect A/D to ring buffer
    { to four channels }
    { as described here }
    { triggering mode }
    { status returned here}
  { call procedure on error }

```

## LSI-11 ANALOG SYSTEM INTERFACE

```

dummy.index := 0;                                { Null queue semaphore
                                                    structure_id; we don't
                                                    want to be signaled }

Start_rtclock (
    source := kwv_100Hz,
    counts := 10,                                { Set the rate for the
                                                    clock which triggers the
                                                    A/D conversions }

    single := FALSE,                             { Base frequency of 100 Hz }

    start := immediate,                         { Ten ticks before
                                                    conversion, i.e. 10 Hz }

    signals := FALSE,                           { Run forever or until we
                                                    stop it }

    timer := dummy,                            { Immediate start }

    state := clk_status );                     { No signaling }

IF clk_status.error_code
    <> IESNORMAL
THEN punt;

WHILE TRUE DO
BEGIN
    GET_ELEMENT (
        LENGTH := 10,                            { Not used, a place holder }

        DATA := analog_data,
        DESC := ring_buf );                   { Error check }

IF sample.status <> IESNORMAL
THEN punt;

. . . { Process the buffer of data }

. . .

END;

```

### G.4.4 Asynchronous, Block-Mode A/D Conversion

```

%INCLUDE 'mpp-lib:IOPKTS'
%INCLUDE 'mpp-lib:RHSDSC'
%INCLUDE 'mpp-lib:KVVINC'

TYPE
    sample = ARRAY [1..2] of INTEGER;          { A single sample of 2
                                                    channels of data }

    buffer = ARRAY [1..128] of sample;         { A block of data }

VAR
    ok : BOOLEAN;                            { Indicates initialization
                                                    success or failure }

    buff1,
    buff2, : buffer;                        { Buffers for data }

    index : INTEGER;                         { FOR loop index }

    chan_data : ad_control_type;            { Control structure
                                                    specifying the number of
                                                    channels and which
                                                    channels to sample }

    rd_status : IO$STATUS;                  { State of conversion upon
                                                    return }

    final_status : IO$REPLY_PKT;             { Reply packed with
                                                    completion status }

```

## LSI-11 ANALOG SYSTEM INTERFACE

```

buff_done : structure_desc;           { Queue semaphore which is
                                         signaled when a buffer is
                                         filled }

clk_status : IO$STATUS;             { Status for start_clock }
dummy : STRUCTURE_ID;              { Null Queue semaphore ID }

BEGIN
  WITH chan_data DO
    BEGIN
      num_chan := 2;                  { Number of channels to
                                         sample at each time
                                         interval }

      chan_ctrl[1].chan_num := 0;      { First channel sampled is
                                         channel 0 }

      chan_ctrl[1].gain_sel := ad_gain_1; { Set gain }
      chan_ctrl[2].chan_num := 1;      { Second channel ... }

      chan_ctrl[2].gain_sel := ad_gain_1; { Set gain }

    END;
    ok := CREATE_QUEUE_SEMAPHORE ( { The block completion
                                         semaphore }

      PROCESS_ORDER := FIFO,
      PACKET_ORDER := FIFO,
      DESC := buff_done );          { First in, first out }

    IF NOT ok THEN punt;            { ditto }

    dummy.index := 0;               { Reply here }

    Read_analog_signal ( { Call procedure if error
                           creating queue semaphore }

      buffer := buff1,                { Null signal semaphore, not
                                         used }

      number := 128,                 { Start A/D conversion for
                                         the first buffer }

      chan_ptr := ADDRESS(chan_data), { samples per buffer }

      trigger := real_time_clock,    { number of channels and
                                         which channels described
                                         here }

      reply := buff_done );          { triggering mode }

    { Reply semaphore }

  Start_rtclock ( { Set the rate for the clock
                     which triggers the A/D
                     conversions }

    source := kwv_1kHz,             { Base frequency of 1000 Hz}

    counts := 2,                   { Ticks before conversion,
                                   i.e. 500 Hz }

    single := FALSE,               { Run forever or until we
                                   stop it }

    start := event,                { Externally triggered start }

    signals := FALSE,              { No signaling }

    timer := dummy,                { Not used }

    state := clk_status );        { Error check }

  IF clk_status.error_code
    <> IE$NORMAL
  THEN punt;

FOR index := 1 TO 10 DO
BEGIN
  Read_analog_signal ( { If problems with the
                           start clock call, then
                           quit }

    buffer := buff2,                { Collect 20 blocks of data}

    number := 128,                 { Start A/D conversion for
                                         the second buffer }

    chan_ptr := ADDRESS(chan_data), { samples per buffer }

    trigger := real_time_clock,    { number of channels and
                                         which channels described
                                         here }

    reply := buff_done );          { triggering mode }

    { with status returned here}

```

## LSI-11 ANALOG SYSTEM INTERFACE

```
RECEIVE (
    VAL_DATA := final_status,
    VAL_LENGTH :=
        SIZE(final_status),
    DESC := buff_done );
IF final_status.status.error_code
    <> success THEN punt;
    { call procedure on error }

.
. { Write first buffer of data }
.

Read_analog_signal
    ( buffer := buff1,
    number := 128,
    chan_ptr := ADDRESS(chan_data),
    trigger := real_time_clock,
    reply := buff_done );
    { Start A/D conversion for
    { the first buffer again}
    { samples per buffer }
    { number of channels and
    { which channels described
    here }
    { triggering mode }
    { with status returned here}

RECEIVE (
    VAL_DATA := final_status,
    VAL_LENGTH :=
        SIZE(final_status),
    DESC := buff_done );
IF final_status.status.error_code <> IESNORMAL
    THEN punt;
    { call procedure on error }

.
. { Write second buffer of data }
.

END; { End of FOR index ... loop }
Stop_rtclock; { All done, turn clock off }
END. { End of module }
```

APPENDIX H  
KXT11-C PERIPHERAL PROCESSOR INTERFACE

The MicroPower/Pascal interface to the KXT11-C Peripheral Processor consists of the following:

- KXT11-C Two-Port RAM (KX and KK) handlers -- KX for arbiter side, KK for KXT11-C side
- KXT11-C TU58 (DD) device handler
- KXT11-C DMA Transfer Controller (QD) handler
- KXT11-C Asynchronous Serial Line (XL) handler
- KXT11-C Synchronous Serial Line (XS) handler
- KXT11-C Parallel Port and Timer/Counter (YK) handler
- Line-Frequency Clock (CK) handler
- Arbiter side Two-Port RAM functions `KX_write_data` and `KX_read_data`, which use the KX handler
- KXT11-C side Two-Port RAM functions `KK_write_data` and `KK_read_data`, which use the KK handler
- DMA Transfer Controller functions `SDMA_TRANSFER`, `SDMA_SEARCH`, `SDMA_SEARCH_TRANSFER`, `SDMA_ALLOCATE`, `SDMA_DEALLOCATE`, and `SDMA_GET_STATUS`, which use the QD handler
- Parallel port and timer/counter functions `YK_PORT_READ`, `YK_PORT_WRITE`, `YK_SET_PATTERN`, `YK_SET_TIMER`, `YK_READ_TIMER`, and `YK_CLEAR_TIMER`, which use the YK handler
- Application load procedure `KXT_LOAD`

The sections that follow describe the components of the MicroPower/Pascal interface to the KXT11-C Peripheral Processor. Sections H.1 and H.2 describe the interface to the KX and KK handlers, which together implement the arbiter/KXT11-C protocol. Section H.3 describes the KXT11-C TU58 interface; Section H.4, the DMA I/O interface; Section H.5, the asynchronous serial line interface; Section H.6, the synchronous serial line interface; Section H.7, the parallel I/O (PIO) and timer/counter interface; and Section H.8, the line-frequency clock interface.

Section H.9 describes the interface for loading procedures onto the KXT11-C.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The following files are referred to throughout this appendix:

RHSLIB.OBJ	Real-time I/O library
KXINC.PAS	Arbiter side Two-Port RAM functions include file
KKINC.PAS	KXT11-C side Two-Port RAM functions include file
QDINC.PAS	DMA Transfer Controller functions include file
YKINC.PAS	Parallel port and timer/counter functions include file
IODEF.PAS	Pascal device I/O include file
KXLINC.PAS	KXT11-C load procedure include file

The RHSLIB object library contains the object modules for the Two-Port RAM, DMA Transfer Controller, and parallel port functions listed at the beginning of this appendix, as well as the object module for the KXT\_LOAD procedure. (Appendix G describes other MicroPower/Pascal procedures that are contained in the RHSLIB library.)

The KXINC include file externally declares the arbiter side Two-Port RAM functions; KKINC, the KXT11-C side Two-Port RAM functions; QDINC, the DMA Transfer Controller functions; and YKINC, the parallel port and timer/counter functions. Each include file also defines data structures associated with the declared functions.

The IODEF include file defines device characteristics parameters, error and severity codes, function codes, and function-independent modifier bits for Pascal device I/O requests.

### NOTE

The KKINC, KKINC, QDINC, and YKINC include files use definitions from the IODEF include file. Therefore, you must include IODEF.PAS in any module that includes KXINC, KKINC, QDINC, or YKINC.

The KXLINC include file externally declares the KXT\_LOAD procedure and defines a data structure associated with it.

### H.1 ARBITER INTERFACE TO THE KXT11-C

The MicroPower/Pascal interface to the arbiter side of the arbiter/KXT11-C protocol consists of the following:

- KX Two-Port RAM handler
- KX\_write\_data function
- KX\_read\_data function

The KX handler provides a standard device handler interface to the arbiter side of the arbiter/KXT11-C protocol. See the KX handler section of Chapter 4 for a detailed description of that interface.

The sections that follow describe the MicroPower/Pascal functions that are used to communicate with the KX handler. Each function allocates an I/O packet, fills it with information based on the function parameters, and sends it to the handler.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

If a reply semaphore is provided in the function call, the function returns immediately after sending the handler request. When the operation is complete, the KX handler sends a standard device handler reply via the specified semaphore. (The handler reply is described in Section 4.1.3.) 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 will wait for the KX handler reply before returning to the caller.

The file KXINC.PAS defines the function-level interface and may be included by any Pascal module that needs to send/receive data via the KX driver.

The following data structure is referenced throughout the rest of this section:

```
TYPE  
  SKX_unit = 0..1; { unit numbers for KXT11-Cs }
```

See the KX handler section in Chapter 4 for more information about KXT11-C unit numbers.

### H.1.1 Sending Data to the KXT11-C

The KX\_write\_data function transfers data from an arbiter buffer to a KXT11-C process and returns a completion-status value of type IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 for a list of completion-status values.

The syntax for calling this function is as follows:

```
KX_write_data ( buffer,length,ret_length,controler,unit,reply,  
                 nowait,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
controler	CHAR	Optional controller (KXT11-C) designation letter; default is 'A'
unit	SKX_unit*	Optional unit number specifying the unit to send the data to; default is 0
reply	STRUCTURE_DSC TR	Optional pointer to reply semaphore descriptor; default is NIL, which will cause function to create necessary semaphore for user, then delete it at end of function call

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

nowait	BOOLEAN	Optional retry parameter; TRUE inhibits retries, FALSE (the default) allows retries
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in handler reply packet; default is 0 (0 is returned in reply packet)

You supply the data to be sent through the dual-port registers in the buffer parameter and the length of the data in the length parameter. Those parameters must always be specified.

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 handler reply packet.

### H.1.2 Receiving Data from the KXT11-C

The KX\_read\_data function transfers data from a KXT11-C process to an arbiter buffer and returns a completion-status value of type IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 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,
                nowait,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-C) designation letter; default is 'A'
unit	SKX_unit	Optional unit number specifying the unit to send requests to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to reply semaphore descriptor; default is NIL, which will cause function to create necessary semaphore for user, then delete it at end of function call
nowait	BOOLEAN	Optional retry parameter; TRUE inhibits retries, FALSE (the default) allows retries

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in handler reply packet; default is 0 (0 is returned in reply packet)
---------	----------	---

You supply the place for the data to go in the buffer parameter and the length of the data in the length parameter. Those parameters must always be specified.

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 handler reply packet.

### H.2 KXT11-C INTERFACE TO THE ARBITER

The MicroPower/Pascal interface to the KXT11-C side of the arbiter/KXT11-C protocol consists of the following:

- KK Two-Port RAM handler
- KK\_write\_data function
- KK\_read\_data function

The KK handler provides a standard device handler interface to the KXT11-C side of the arbiter/KXT11-C protocol. See the KK handler section of Chapter 4 for a detailed description of that interface.

The sections that follow describe the MicroPower/Pascal functions that are used to communicate with the KK handler. Each function allocates an I/O packet, fills it with information based on the function parameters, and sends it to the handler.

If a reply semaphore is provided in the function call, the function returns immediately after sending the handler request. When the operation is complete, the KK handler sends a standard device handler reply via the specified semaphore. (The handler reply is described in Section 4.1.3.) The completion status and 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 KK handler reply before returning to the caller.

The file KKINC.PAS defines the function-level interface and may be included by any Pascal module that needs to send/receive data via the KK handler.

The following data structure is referenced throughout the rest of this section:

```
TYPE
  SKK_unit = 0..1; { unit numbers }
```

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

### 4.2.1 Receiving Data from the LSI-11 Bus

The KK\_read\_data function transfers data from the arbiter to a KXT11-C buffer and returns a completion-status value of type IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 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	SKK_unit	Optional unit number specifying unit the request is to be sent to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to reply semaphore descriptor; default is NIL
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in handler 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 handler reply packet.

### H.2.2 Sending Data to the LSI-11 Bus

The KK\_write\_data function transfers data from a KXT11-C buffer to the arbiter and returns a completion-status value of type IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 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
VAR ret_length	UNSIGNED	Variable that returns number of bytes actually transferred -- not returned if reply parameter provided

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

unit	SYK_unit	Optional unit number specifying unit that data is to be sent to; default is 0
reply	STRUCTURE_DESC_PTR	Optional pointer to reply semaphore descriptor; default is NIL
seq_num	UNSIGNED	Optional user-defined word value, returned unmodified in handler 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 handler reply packet.

### H.3 KXT11-C TU58 INTERFACE

The KXT11-C DD handler provides a standard device handler interface to a TU58 cartridge tape subsystem connected to a KXT11-C. See the two DD Handler sections of Chapter 4 for descriptions of that interface.

### H.4 KXT11-C DMA TRANSFER CONTROLLER INTERFACE

The MicroPower/Pascal interface to the DMA Transfer Controller (DTC) on the KXT11-C consists of the following:

- QD DMA Transfer Controller handler
- SDMA\_TRANSFER function
- SDMA\_SEARCH function
- SDMA\_SEARCH\_TRANSFER function
- SDMA\_ALLOCATE function
- SDMA\_DEALLOCATE function
- SDMA\_GET\_STATUS function

The QD handler provides a standard device handler interface to the 2-channel DMA controller on the KXT11-C. See the QD handler section of Chapter 4 for a description of that interface.

The sections that follow describe the MicroPower/Pascal functions that are used to communicate with the QD device handler. Each function allocates an I/O packet, fills it with information based on the function parameters, and sends it to the handler.

If a reply semaphore is provided in a transfer, search, or Get Status function call, the function returns immediately after sending the handler request. When the operation is complete, the QD handler sends a standard device handler reply via the specified semaphore. (The handler reply is described in Section 4.1.3.) 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

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

routine that waits on the semaphore must also process the actual-length information in the packet.

If no reply semaphore is provided in a transfer, search, or Get Status function call or if the function called was SDMA\_ALLOCATE or SDMA DEALLOCATE, the function waits for the QD handler reply before returning to the caller.

The file QDINC.PAS defines the function-level interface and may be included by any Pascal module that needs to communicate with the 2-channel DMA Transfer Controller.

The following data structures are referenced throughout the rest of this section:

### TYPE

```
DMASADDR_SPACE = (DMASIBUS, DMASQBUS);      { local or qbus space }

DMASINCR_OPTION = (DMASUP, DMASDOWN, DMASNOINC); { increment up,
                                                       down, or not at all }

DMASWAIT_OPTION = (DMASWAIT_0, DMASWAIT_1,
                    DMASWAIT_2, DMASWAIT_4); { add 0, 1, 2, or 4
                                              wait states }

DMASREQ_OPTION = (DMASNOWFR, DMASNOI); { wait for request line
                                           active or not }

DMASIO_OPTION = (DMASNOIO, DMASIO); { access I/O addresses or
                                         not }

DMASBYTE_OPTION = (DMASNOBYTE, DMASBYTE); { byte mode to/from this
                                             address or not }

DMASADDRESS = PACKED RECORD
  LOW: [POS(00),WORD] UNSIGNED; { source or destination
                                dma address }
  HIGH: [POS(16),BYTE] 0..63; { low portion of 22bit
                               address }
  IO: [POS(24),BIT(1)] DMASIO_OPTION; { high portion of 22bit
                                         address }
  WS: [POS(25),BIT(2)] DMASWAIT_OPTION; { IO mode? }
  INC: [POS(27),BIT(2)] DMASINCR_OPTION; { number of wait states
                                             to add }
  WFR: [POS(29),BIT(1)] DMASREQ_OPTION; { UP or DOWN or NOINC }
  BM: [POS(30),BIT(1)] DMASBYTE_OPTION; { wait for request? }
  SPACE: [POS(31),BIT(1)] DMASADDR_SPACE; { byte mode? }
END; { IBUS OR QBUS }

DMASBYTE_COUNT = UNSIGNED; { number of bytes to
                           transfer }

DMASUNIT_NUMBER = 0..1; { 0 = channel A, 1 = channel B }

DMASSEM_POINTER = ^ SEMAPHORE_DESC; { pointer to sem desc }

DMASDEVICE_REGS = PACKED RECORD { Device registers }
  caoff_b_1 : unsigned; {Current address req, offset, B,
                        ch. 1}
  caoff_b_0 : unsigned; {Current address req, offset, B,
                        ch. 0}
```

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

```

baoff_b_1 : unsigned;           {Base address req, offset, B, ch.
}                                1}
baoff_b_0 : unsigned;           {Base address req, offset, B, ch.
}                                0}
caoff_a_1 : unsigned;           {Current address req, offset, A,
}                                ch. 1}
caoff_a_0 : unsigned;           {Current address req, offset, A,
}                                ch. 0}
baoff_a_1 : unsigned;           {Base address req, offset, A, ch.
}                                1}
baoff_a_0 : unsigned;           {Base address req, offset, A, ch.
}                                0}
cataq_b_1 : unsigned;           {Current address req, set tail, B,
}                                ch. 1}
cataq_b_0 : unsigned;           {Current address req, set tail, B,
}                                ch. 0}
bataq_b_1 : unsigned;           {Base address req, set tail, B, ch.
}                                1}
bataq_b_0 : unsigned;           {Base address req, set tail, B, ch.
}                                0}
cataq_a_1 : unsigned;           {Current address req, set tail, A,
}                                ch. 1}
cataq_a_0 : unsigned;           {Current address req, set tail, A,
}                                ch. 0}
bataq_a_1 : unsigned;           {Base address req, set tail, A, ch.
}                                1}
bataq_a_0 : unsigned;           {Base address req, set tail, A, ch.
}                                0}
chaino_1 : unsigned;            {Chain load address req, offset,
}                                ch. 1}
chaino_0 : unsigned;            {Chain load address req, offset,
}                                ch. 0}
chaint_1 : unsigned;             {Chain load address req, set tail,
}                                ch. 1}
chaint_0 : unsigned;             {Chain load address req, set tail,
}                                ch. 0}
isr_1 : unsigned;                {Interrupt save register, ch. 1}
isr_0 : unsigned;                {Interrupt save register, ch. 0}
stat_1 : unsigned;                {Command/Status register, ch. 1}
stat_0 : unsigned;                {Command/Status register, ch. 0}
coc_1 : unsigned;                {Current operation count, ch. 1}
coc_0 : unsigned;                {Current operation count, ch. 0}
boc_1 : unsigned;                {Base operation count, ch. 1}
boc_0 : unsigned;                {Base operation count, ch. 0}
mmr : unsigned;                  {Master mode register}

junk : [WORD(7)];               {Pattern register, ch. 1}
pat_1 : unsigned;                {Pattern register, ch. 0}
pat_0 : unsigned;                {Mask register, ch. 1}
msk_1 : unsigned;                {Mask register, ch. 0}
cmr_l_1 : unsigned;              {Channel mode register, low, ch.
}                                1}
cmr_l_0 : unsigned;              {Channel mode register, low, ch.
}                                0}
cmr_h_1 : unsigned;              {Channel mode register, high, ch.
}                                1}
cmr_h_0 : unsigned;              {Channel mode register, high, ch.
}                                0}
inv_1 : unsigned;                {Interrupt vector register, ch. 1}
inv_0 : unsigned;                {Interrupt vector register, ch. 0}

END;

CONST
{

```

## KXTII-C PERIPHERAL PROCESSOR INTERFACE

These constants are used to initialize variables of type DMASADDRESS.

```
} DMASNORM_IBUS_ADDRESS = DMASADDRESS (0, 0, DMAHIGH, DMAHIGH,
                                         DMAHUP, DMAHOWER, DMAHOWER,
                                         DMAIBUS );
DMASNORM_QBUS_ADDRESS = DMASADDRESS (0, 0, DMAHIGH, DMAHIGH,
                                         DMAHUP, DMAHOWER, DMAHOWER,
                                         DMAQBUS );
```

{ Constant for the size of the register file }

DMASRECBUF\_SIZE = 92;

### H.4.1 DMA Transfer

The SDMA\_TRANSFER function transfers data from one place to another. The function returns a value of type DMASBYTE\_COUNT -- the amount of bytes transferred (0 if an error occurred or REPLY parameter was provided).

The syntax for calling this function is as follows:

SDMA\_TRANSFER ( SOURCE, DEST, COUNT, UNIT, REPLY )

Parameter	Type	Description
SOURCE	DMASADDRESS	Address -- either a bus local -- from which data will be transferred
DEST	DMASADDRESS	Address -- either a bus local -- to which data will be transferred
COUNT	DMASBYTE_COUNT	Number of bytes to transfer
UNIT	DM_SUN_NUMBER	Optional unit number; default is 0 (channel A)
REPLY	DMASSEM_POINTER	Optional pointer to semaphore descriptor; default is NULL, which will cause function to create necessary semaphore for user, then delete it at end of function call

Either SOURCE or DEST must specify a local buffer address. If 'SRC' is a buffer, data will be read from a KXTII-C or LSI-11-bus address to the buffer. If SOURCE is a buffer, data will be written from the buffer to a KXTII-C or LSI-11-bus address.

### H.4.2 DMA Search

The SDMA\_SEARCH function searches a specified source area. The search terminates either when the search descriptor is matched or when a specified byte count expires. This function returns a value of type DMASBYTE\_COUNT -- the number of bytes searched (0 if an error occurred or REPLY parameter provided).

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The syntax for calling this function is as follows:

```
SDMA_SEARCH ( SOURCE,COUNT,VAL,MASK,UNIT,REPLY )
```

Parameter	Type	Description
SOURCE	DMASADDRESS	Address -- either Q-BUS or local -- at which search will begin
COUNT	DMASBYTE_COUNT	Maximum number of bytes to search
VAL	UNSIGNED	Search value
MASK	UNSIGNED	Optional search mask; default is 0
UNIT	DMASUNIT_NUMBER	Optional unit number; default is 0 (channel A)
REPLY	DMASSEM_POINTER	Optional pointer to semaphore descriptor; default is NIL, which will cause function to create necessary semaphore for user, then delete it at end of function call

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 the high-order eight bits all set to 1. Thus, the low-order byte of each word in the buffer will be compared with the low-order byte of the VAL 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 searching the high-order byte, 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 VAL and MASK parameters would be, for the low-order search, VAL := value\_to\_find and MASK := \$0'177400', and, for the high-order search, VAL := (Value\_to\_find \* 256) and MASK := \$0'377'.

### H.4.3 DMA Transfer While Search

The DMA\_SEARCH\_TRANSFER function causes data to be transferred from one place to another until either a search value is matched or a byte count expires. This function returns a value of type DMASBYTE\_COUNT -- the number of bytes transferred (0 if an error occurred or REPLY parameter provided).

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The syntax for calling this function is as follows:

```
SDMA_SEARCH_TRANSFER ( SOURCE, DEST, COUNT, VAL, MASK, UNIT, REPLY )
```

Parameter	Type	Description
SOURCE	DMASADDRESS	Address -- either Q-BUS or local -- at which search will begin and from which data will be transferred
DEST	DMASADDRESS	Address -- either Q-BUS or local -- to which data will be transferred
COUNT	DMASBYTE_COUNT	Maximum number of bytes to transfer
VAL	UNSIGNED	Search value
MASK	UNSIGNED	Optional search mask; default is 0
UNIT	DMASUNIT_NUMBER	Optional unit number; default is 0 (channel A)
REPLY	DMASSEM_POINTER	Optional pointer to semaphore descriptor; default is NIL, which will cause function to create necessary semaphore for user, then delete it at end of function call

Either SOURCE or DEST must specify a local buffer address. If DEST is a buffer, data will be read from a KXT11-C address or an LSI-11-bus address to the buffer. If SOURCE is a buffer, data will be written from the buffer to a KXT11-C address or an LSI-11-bus address.

### H.4.4 DMA Channel Allocation

The SDMA\_ALLOCATE function allocates a specified unit for the exclusive use of the calling process. This function returns a Boolean value indicating success (TRUE) or failure (FALSE).

The syntax for calling this function is as follows:

```
SDMA_ALLOCATE ( UNIT )
```

Parameter	Type	Description
UNIT	DMASUNIT_NUMBER	Optional unit number; default is 0 (channel A)

### H.4.5 DMA Channel Deallocation

The SDMA\_DEALLOCATE function reverses the effect of a previous SDMA\_ALLOCATE call. This function returns a Boolean value indicating success (TRUE) or failure (FALSE).

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The syntax for calling this function is as follows:

`SDMA_DEALLOCATE ( UNIT )`

Parameter	Type	Description
UNIT	DMASUNIT_NUMBER	Optional unit number; default is 0 (channel A)

### H.4.6 DMA Status Return

The `SDMA_GET_STATUS` function returns status information -- the contents of device registers -- from the specified DMA unit into a user-supplied buffer. This function returns a Boolean value indicating success (TRUE) or failure (FALSE).

The syntax for calling this function is as follows:

`SDMA_GET_STATUS ( UNIT, REGBUF, REGBUF_SIZE, REPLY )`

Parameter	Type	Description
UNIT	DMASUNIT_NUMBER	Optional unit number; default is 0 (channel A)
VAR REGBUF	DMASDEVICE_REGS	Buffer to which status information is to be returned; data type DMASDEVICE_REGS, listed above, shows format of device register information that is returned
REGBUF_SIZE	INTEGER	Optional buffer size; default is DMASREGBUF_SIZE (92 bytes)
REPLY	DMASSEM_POINTER	Optional pointer to reply semaphore descriptor; default is NIL, which will cause function to create necessary semaphore for user, then delete it at end of function call

### H.4.7 DMA Function Call Example

Shown below is a simple test program in which a process calls the `SDMA_TRANSFER` and `SDMA_SEARCH` functions. Note the use of the QDINC include file to define the functions and their associated data types. This example is intended only to illustrate the function calls. Only the calling process is shown.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

```
PROGRAM test;
{$include 'iodef.pas'
{$include 'qdinc.pas'

VAR
    a,b : DMA$ADDRESS;
    d : SEMAPHORE DESC;
    e : PACKED ARRAY[0..511] OF CHAR;

BEGIN
    a := DMASNORM_IBUS_ADDRESS;           { init a to
                                             defaults }
    b := DMASNORM_QBUS_ADDRESS;           { init b to
                                             defaults }

    a.LOW := (ADDRESS(e)) :: UNSIGNED;     { local address }
    b.LOW := 12346;                        { some QBUS
                                             address }
    b.HIGH := 23;                          { some QBUS
                                             address }

    SDMA_TRANSFER (
        SOURCE := a,
        DEST := b,
        COUNT:= SIZE(e));

    if (SDMA_SEARCH ( a , SIZE(e) , ('$') :: UNSIGNED .. 1 ) < SIZE(e) ) then
        SDMA_TRANSFER( a, b, 10,
                       REPLY := address(d) );
END.
```

### H.5 KXT11-C ASYNCHRONOUS SERIAL LINE INTERFACE

The KXT11-C XL handler provides a standard device handler interface to devices connected to any of the three serial I/O ports on the KXT11-C. See the KXT11-C XL handler section of Chapter 4 for a detailed description of this interface.

### H.6 KXT11-C SYNCHRONOUS SERIAL LINE INTERFACE

The KXT11-C XS handler provides a standard device handler interface to the KXT11-C multiprotocol chip SLU2 Channel A for synchronous serial I/O. See the XS handler section of Chapter 4 for a detailed description of this interface.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

### H.7 KXT11-C PARALLEL PORT AND TIMER/COUNTER INTERFACE

The MicroPower/Pascal interface to the PIO and timer/counter hardware on the KXT11-C consists of the following:

- YK\_parallel\_port\_and\_timer/counter\_handler
- 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 YK handler provides a standard device handler interface to the PIO and timer/counter hardware on the KXT11-C. See the YK handler section of Chapter 4 for a description of this interface.

The sections that follow describe the MicroPower/Pascal functions that are used to communicate with the YK device handler. Each function allocates an I/O packet, fills it in with information based on the function parameters, and sends it to the handler.

If a reply semaphore is provided in the function call, the function returns immediately after sending the handler request. When the operation is complete, the YK handler sends a standard device handler reply via the specified semaphore. (The handler reply is described in Section 4.1.3.) The completion status returned in the reply packet must be processed by a routine that is waiting on the semaphore. For PIO read or 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 YK handler reply before returning to the caller.

The file YKINC.PAS defines the function-level interface and may be included by any Pascal module that needs to communicate with the KXT11-C parallel ports or timer/counters.

The YK handler and the functions listed above allow you to issue multiple requests for a single KXT11-C 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 requestor.

In addition, pattern-matching commands can be issued in conjunction with the 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 second, different special character is received. The function calls to accomplish that 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 handler can continue receiving the second portion while you are processing the first.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The following data structures, defined in YKINC.PAS, are referenced throughout the rest of this section:

```

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 }

  Port_mods = (                 { Port's Function Modifiers }
    nu_1,                        { - not used }
    nu_2,                        { - not used }
    reset_pat );                { - reset pattern at end }

  Port_mod_entry = PACKED SET OF Port_mods;

  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 till match mode }

  Pat_mod_entry = PACKED SET OF Pattern_mods;

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

  Timer_mod_entry = PACKED SET OF Timer_mods;

  YKBUF_P_T = ^ UNSIGNED;       { Data buffer pointer }

```

### H.7.1 Reading from a PIO Port

The YK\_PORT\_READ function transfers data from a parallel port to a KXT11-C buffer and returns a completion-status value of type IOSESTATIC (defined in the IODEF include file). See Section 4.1.1 for a list of completion-status values.

The syntax for calling this function is as follows:

```

YK_PORT_READ ( PORT_NUM,BUFFER,BYTE_COUNT,REPLY,MATCH_RST,
                SEQ_NIM )

```

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.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

**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 REPLY is provided.

**REPLY**

**STRUCTURE\_DESC\_PTR**

Optional pointer to a reply 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 handler 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 handler reply packet.

### 8.7.2 Writing to a PIO Port

The YK\_PORT\_WRITE function transfers data from a KXT11-C buffer to a parallel port and returns a completion-status value of type IOSTATUS (defined in the IODEF include file). See Section 4.1.1 for a list of completion-status values.

The syntax for calling this function is as follows:

```
YK_PORT_WRITE ( PORT_NUM,BUFFER,BYTE_COUNT,REPLY,MATCH_RST,
                 SEQ_NUM )
```

Parameter	Type	Description
<b>PORT_NUM</b>	<b>UNIT_NUMBPP</b>	Port number to be written to.
<b>VAR BUFFER</b>	<b>UNIVERSAL</b>	Data buffer address.
<b>VAR BYTE_COUNT UNSIGNED</b>		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

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

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 REPLY was provided.

REPLY	STRUCTURE_DESC_PTR	Optional pointer to a reply 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 handler 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 handler reply packet.

### H.7.3 Pattern Recognition

The YK\_SET\_PATTERN function 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 the 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 IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 for a list of completion-status values.

The syntax for calling this 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

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

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 a reply semaphore descriptor; default is NIL.
PATP	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.
PATT	BYTE_RANGE	
PATM	BYTE_RANGE	
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 handler reply packet.
SEQ_NUM	UNSIGNED	Optional user-defined word value, returned unmodified in handler reply packet; default is 0 (0 is returned in reply packet)

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

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

### 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 := $0'17',  
                 patt := $0'200',  
                 patm := $0'357')
```

### H.7.4 PIO DMA Process

If you want to perform DMA transfers, a separate process must be built into the application. That process must first allocate a packet and send a DMA read or a DMA write request to the YK handler (see the YK section of Chapter 4) and wait for the reply. If the reply indicates normal status, the process then sends a DMA transfer command to the DMA (QD) handler; otherwise, it reports a software exception. The process 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, the process sends a DMA Complete request to the YK handler, which unlocks the queue of requests for that port.

### H.7.5 Timer/Counter Commands

The functions described in this section are provided to control the timer/counters on the KXT11-C. Each function returns a completion-status value of type IOSSTATUS (defined in the IODEF include file). See Section 4.1.3 for a list of completion-status values.

**H.7.5.1 Timer Setup** - The YK\_SET\_TIMER function can set a timer to an initial value, trigger a Timer after setting it, or set a timer to periodically signal a binary semaphore. This function can be used in conjunction with the YK\_READ\_TIMER function to time or count real-time events.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

The syntax for calling this 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 timer to signal a binary semaphore and restart after each time-out; if timer mode omitted, the mode set by the prefix file or the last timer command remains in effect
REPLY	STRUCTURE_DESC_PTR	Optional pointer to a reply semaphore descriptor; default is NIL
BIN_SEM	STRUCTURE_DESC_PTR	Optional pointer to a continuous-cycle semaphore to be signaled on each timer timeout; default is NIL

H.7.5.2 Reading a Timer - The YK\_READ\_TIMER function reads the current count from a timer/counter.

The syntax for calling this 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 handler reply packet
REPLY	STRUCTURE_DESC_PTR	Optional pointer to a reply semaphore descriptor; default is NIL

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

H.7.5.3 Clearing a Timer - The YK\_CLEAR\_TIMER function deactivates a timer/counter.

The syntax for calling this 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 a reply semaphore descriptor; default is NIL

H.7.5.4 Notes on Timer Requests - The following notes and restrictions apply when using the timer/counters:

1. 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 gets disabled during the setting of timer 3's constant, and therefore the handshake signals also get disabled.
2. When a timer is used in continuous mode, only the binary or counting timer semaphore gets signaled on timeouts. The standard reply semaphore is used only to return a standard device handler reply packet (see Section 4.1.2) indicating whether the timer setup was successful.
3. The timer clear command is used only to clear a timer operation in the continuous mode.

## H.8 LINE-FREQUENCY CLOCK INTERFACE

The MicroPower/Pascal line-frequency clock handler provides a reply/request interface to a clock on the target processor. The clock handler can be used in both KXT11-C and non-KXT11-C applications. The procedure for building the clock-handler process into your application is described in the MicroPower/Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide.

See Chapter 6 for detailed descriptions of the clock handler process, the services it provides, and the request/reply interface to the handler.

## H.9 LOAD APPLICATION ONTO KXT11-C PROCEDURE

KXT\_LOAD is a MicroPower/Pascal procedure that runs on the arbiter and is used to load and start a MicroPower/Pascal application on a KXT11-C. The procedure reads in a MIM file, one block at a time, and commands a KXT11-C to DMA each block into its local memory. When all blocks have been loaded into the KXT11-C's memory, the QBUS OJT command is given to the KXT11-C, thereby causing the KXT11-C to jump to the address contained in location 24(octal).

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

### H.9.1 MIM File

The file that the KXT\_LOAD procedure loads is the memory image file that is output by the MIM utility. To be loaded onto a KXT11-C with KXT\_LOAD, the memory image must be in an unmapped format and must not contain the Debug Service Module (DSM).

The MIM file may or may not have the T158 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 T158 tape and booted via the KXT11-C console port, or it can reside on one of the arbiter's file-structured devices and be booted via the KXT\_LOAD procedure.

### H.9.2 User's Interface

You gain access to the KXT\_LOAD procedure by defining it as an external procedure and then merging with the RHSLIB, file system, and driver object libraries (in addition to the files that are normally merged with).

The KXT\_LOAD procedure causes requests to be generated to the MicroPower/Pascal Directory Service Process (DSP). Thus, the DSP must be built into your application. You build the DSP into a process by assembling the file DSPPFX.MAC and merging with a DSP object library as described in the MicroPower Pascal-RT System User's Guide and the MicroPower/Pascal-RSX/VMS System User's Guide. Note that the KXT\_LOAD procedure uses only the Open, Find, Read, and Close operations; if you want to optimize your application, you may delete all other functions from a copy of DSPPFX.MAC.

The KXT\_LOAD procedure must be merged with the mapped or unmapped driver object library. At the end of the list of modules being merged -- after RHSLIB and the appropriate LOC module library (LTHXX -- add DRVM for a mapped arbiter or CRVM for an unmapped arbiter).

A process that calls the KXT\_LOAD procedure must observe the following guidelines:

1. The process must have access to the T158 port.
2. The KXT\_LOAD procedure can load only one KXT11-C at a time. If multiple KXT11-C loads are required, the calling process must serialize them.
3. The KXT\_LOAD procedure is active continuously through the load, which prevents any processes at a lower priority from running. If the KXT\_LOAD procedure is to be used during normal operation -- i.e., sometime other than during system initialization -- it should be called by a process with a priority lower than any real-time interruptable processes.

## KXT11-C PERIPHERAL PROCESSOR INTERFACE

You can define the KXT\_LOAD procedure by including the file KXLINK.PAS in your program. The KXLINK include file also defines the completion-status codes that the procedure returns, as follows:

```
TYPE
  KXT_Error_Code = (
    normal,           { normal, no error has occurred }
    kxt_state,        { KXT11-C is in the wrong state }
    non_mim,          { file is not a MIM file }
    mapped_mim,       { MIM file is in a mapped format }
    kxt_addr );      { Attempt to load illegal KXT11-C address }
```

The syntax for calling the procedure is as follows:

```
KXT_LOAD ( KXT_ADDR, MIMF, RFTC );
```

Parameter	Type	Description
KXT_ADDR	UNSIGNED	Virtual address of KXT11-C
VAR MIMF	1READONLY PACKED ARRAY OF CHAR	File name (L..V : INTEGER)
VAR RFTC	KXT_Error_Code	KXT11-C return code

### H.9.3 Program Example

The following program uses the KXT\_LOAD procedure to load a sample KXT11-C application into KXT11-C ID 2. (KXT11-C ID numbers are listed in the KX handler section of Chapter 4.)

```
SYSTEM(MICROPOWER), STACK_SIZE(1000), PRIORITY(10), DEV_ADDRESS )
PROGRAM LOADK;

{ Define the Load Procedure }
{include 'KXLINK.PAS' }

{ Define Address for KXT11-C ID 2 }
CONST
  KXT2_ADDR = $01E62173;

{ Define Error Return Status }
VAR
  LOAD_RESULTS : KXT_Error_Code;

{ Main Program }
BEGIN
  KXT_LOAD (KXT2_ADDR, 'DYAP:EXKXT.MIM', LOAD_RESULTS);
END.
```

## INDEX

AA device handler, 4-17, G-8  
AAV11-C, G-1, G-13  
AB device interface, G-13  
Abort process, 7-7  
Active files, 5-1, 5-5, E-7  
ADV11-C, 4-17, G-1, G-8  
Allocate Packet, 3-12  
ALPC\$, 3-10  
ALPK\$, 3-12  
Analog I/O interface, G-1  
Analog-to-digital (A/D)  
    conversion, 4-17, G-8,  
    G-14.1, G-17, G-18  
Arbiter interface to KXT11-C, H-2  
Argument  
    null, 3-3  
    optional, document convention,  
        xv  
Argument block, primitive service  
    request, 3-1  
Argument list  
    exception, 7-8  
    subroutine call, D-2  
AXV11-C, 4-17, G-1, G-8, G-13

Binary semaphore, 1-5, 2-26  
Block  
    file identification (FIB), 5-1,  
    5-10 to 5-13  
    fork descriptor, 8-10  
    home, E-2  
    interrupt dispatch (IDB), 2-34,  
    8-3  
    process control (PCB), 2-12 to  
    2-16  
    process descriptor (PDB), 3-7  
    to 3-9  
    structure descriptor (SDB), 3-5  
    to 3-7  
    structure name, 2-24  
Block mode, 4-63, 4-105, 5-1,  
    8-2, 8-3  
Blocking, process, 2-9  
Buffer  
    circular, 2-28  
    I/O, 4-6, 4-8, 5-3, 5-12  
    ring, 2-28 to 2-30

Cancel One or More Timer  
    Requests, 6-7

CCNDS, 3-14, 7-7  
Change Process Priority, 3-17  
CHGP\$, 3-17  
CINT\$, 3-19, 8-1  
Circular buffer, 2-28  
Clock process, 1-12, 4-2 to 4-5,  
    4-74, 6-1, H-22  
Codes  
    exception group, 2-11, 2-14,  
    3-15, 3-24, 3-35, 7-2  
    I/O completion status, 4-12 to  
    4-15, 5-2, 9-17  
    I/O function request, 4-6, 5-2  
    to 5-6, 9-15  
    process mapping type, 2-14  
    process state modifier, 2-14  
    RTDSP function request, 5-4, E-20  
Common, system-, memory  
    organization, 2-33, 2-34  
Common data area, 2-3  
Communications line  
    device handler, 4-74  
    protocol, 4-81  
Completion, I/O device, 4-8, 4-12  
    to 4-16, 5-6  
Concurrent file access, 5-4  
Concurrent process, 2-1  
Concurrent programming, 2-1  
Conditional Get Element, 3-44  
Conditional Put Element, 3-54  
Conditional Receive Data, 3-62  
Conditional Send Data, 3-93  
Conditional Wait on Queue  
    Semaphore, 3-107  
Conditionally Allocate Packet, 3-10  
Conditionally Signal Queue  
    Semaphore, 3-91  
Conditionally Signal Semaphore,  
    3-85  
Conditionally Wait on Semaphore,  
    3-101  
Connect to Exception Condition,  
    3-14, 7-7  
Connect to Interrupt, 3-19, 8-1  
Context switch  
    kernel, 2-12  
    kernel primitive, 2-13, 2-15  
    option bits, 2-14, 3-23, 3-25,  
    3-26, 3-34 to 3-36  
PC, 2-15  
process, 2-12  
PS, 2-15  
user, 2-13, 2-15

## INDEX

**Controller**  
 configuration data, device handler, 9-5 to 9-8.2  
 device, 4-3  
 number supported, 9-5, 9-6  
 process, 9-15  
 units per, 4-3, 4-4, 9-5, 9-6  
**Conventions, document**, x/  
**Counting semaphore**, 1-6, 2-27  
**CPU**  
 shared, 2-1  
 virtual, 2-1  
**Create Process**, 3-22  
**Create Structure**, 3-26  
**CRPCS**, 3-22, 9-14  
**CRSTS**, 3-26  
**CTR**, 6-7  
**CTRCE\$ macro**, 9-6 to 9-8.2  
  
**DAPK\$**, 3-29  
**Data segment, process**, 2-2  
**Data structures**, 2-23 to 2-34  
 creation, 2-3, 2-24  
 deletion, 2-24  
 device handler initialization, 9-7 to 9-8.2  
 message packets, 2-30, 2-31, 4-5, 4-12, 6-2  
 system-common memory, organization, 2-33, 2-34  
**system queues**, 2-31  
**typed data structures**, 2-24 to 2-30  
 binary semaphore, 2-26  
 counting semaphore, 2-27  
 name block, 2-24  
 queue semaphore, 2-27  
 ring buffer, 2-28 to 2-30  
 process control block, 2-12 to 2-16  
 unformatted, 2-30  
**DD device handler**, 4-26  
 for KXT11-C, 4-26, 4-92, 9-7, H-7  
**\$DDEXC**, 9-19  
**\$DDINI**, 9-14  
**Deallocate Packet**, 3-29  
**DECtape II device handler**, 4-26  
 for KXT11-C, 4-26, 4-92, 9-7, H-7  
**Define an Impure Program Data Section**, 3-52  
**Define a Pure Program Data Section**, 3-53  
**Define a Pure Program Instruction Section**, 3-58  
**Define Static Process**, 3-33  
**Definition macro, device handler**  
 impure area, 9-9  
**Delete Process**, 3-39  
**Delete Structure**, 3-40

**Device driver.** See **Device handler**  
**Device handler**  
 AA, 4-17, G-8  
 access mapping, 2-19, 2-21  
 ADV11-C/AXV11-C, 4-17, G-8  
 completion, 4-8, 4-12 to 4-16  
 configuration data, 9-5  
 controller process, 9-15  
 custom, 9-1 to 9-20  
 DD, 4-26  
 for KXT11-C, 4-26, 4-92, 9-7, H-7  
 DL, 4-31  
 DU, 4-37  
 DY, 4-43, A-2  
 exceptions, 9-19  
 function codes, 4-6, 5-4  
 functions, 9-1, 9-10, 9-12  
 general interface, 4-2  
 I/O request packet, 4-2, 4-5 to 4-2  
 kernel interface, 9-1, 9-2  
 KK, 4-93, H-5  
 KW, 4-49, G-3  
 KWV11-C, 4-49, G-3  
 KX, 4-55, 9-7, H-2  
**KXT11-C**  
 asynchronous serial line, 4-105, H-14  
 clock, H-22  
**DMA Transfer Controller (DTC)**, 4-96, H-7  
**multiprotocol chip**, 4-92, 4-105, 4-116  
**parallel port and timer/counter**, 4-120, H-15  
**synchronous serial line**, 4-116, H-14  
**timer/counter and parallel port**, 4-120, H-15  
**TU58**, 4-26, 4-92, 9-7, H-7  
**two-port RAM (TPR)**, 4-55, 4-93, H-2, H-5  
**mapping**, 2-19, 2-22, 8-2  
**MICRO/PDP-11 disk**, 4-37  
**modem**. See **XL device handler**  
**MSCP disk-class**, 4-37  
**prefix files**, 4-3 to 4-5, 9-2 to 9-8.2  
**priority assignments**, 9-3, 9-4, 9-15  
**process priorities**, B-3  
**QD**, 4-96, H-7  
**RD51**, 4-37  
**request packet**, 4-5, 9-2  
**request queue**, 4-3 to 4-5, 9-2  
**RL01/RL02**, 4-31  
**RX02**, 4-43  
**RX50**, 4-37  
**source file**, A-1  
 components, 9-10  
 controller process, 9-15 to 9-16

## INDEX

copyright page, 9-10  
 declarations, 9-10 to 9-14  
 error-processing routines, 9-18  
 fork routine, 9-17  
 functional description, 9-12  
 initialization process, 9-14  
 interrupt procedure, 9-15  
 interrupt service routine  
     (ISR), 9-16  
 module header, 9-12  
 queue server, 9-15  
 reply routine, 9-12  
 severity level, 4-13, 4-14  
 status codes, 4-12 to 4-15,  
     9-17  
 termination procedure, 9-18  
     writing, 9-1 to 9-20  
 terminal. See XL device  
     handler  
 TU58 DECTape II, 4-26  
     for KXT11-C, 4-26, 4-92, 9-7,  
         H-7  
 XA, 4-59  
 XL, 4-63, A-19  
     for KXT11-C, 4-105, H-14  
 XP, 4-74  
 XS, 4-116, H-14  
 YA, 4-84, A-43  
 YF, 4-88  
 YK, 4-120, H-15  
 DEVICES macro, 8-6  
 DEXCS, 3-31, 7-7  
 DFSPCS, 3-33, 9-12  
 Digital-to-analog (D/A)  
     conversion, 4-17, G-13,  
         G-16  
 DINTS, 3-37, 8-2  
 Directory service process (DSP),  
     5-1, 5-4, E-1 to E-22, H-23  
 Directory structure, E-1 to E-7  
 Disconnect from Interrupt, 3-37,  
     8-2  
 Dismiss Exception, 3-31, 7-7  
 DL device handler, 4-31  
 DLPCS, 3-39  
 DLSTS, 3-40  
 DLV11, 4-63  
 DLV11-E, 4-63  
 DLV11-F, 4-63  
 DLV11-J, 4-63  
 DMA, 8-2  
 \$DMA\_ALLOCATE (QD) function, H-7,  
     H-12  
 \$DMA DEALLOCATE (QD) function,  
     H-7, H-12  
 \$DMA\_GET\_STATUS (QD) function,  
     H-7, H-13  
 \$DMA\_SEARCH (QD) function, H-7,  
     H-10, H-13  
 \$DMA\_SEARCH TRANSFER (QD)  
     function, H-7, H-11  
 \$DMA\_TRANSFER (QD) function, H-7,  
     H-10, H-13

Document conventions, xv  
 Documents, related, xiv  
 Doubly linked lists, 2-33  
 DPV11, 4-74  
 Driver, device. See Device  
     handler  
 Driver access mapping, 2-19, 2-21  
 Driver mapping, 2-19, 2-22, 8-2  
 Drop CPU Priority, 3-58  
 DP.XXX, 4-5 to 4-8, 4-12 to 4-15  
 DRV11, 4-84  
 DRV11-J, 4-59  
 DRVCF\$ macro, 9-5 to 9-6  
 DRVDEF.MAC driver macro library,  
     4-2, 4-100, 9-9  
 DRVDEF\$ kernel macro, 4-2, 4-100  
 DU device handler, 4-37  
 DY device handler, 4-43, A-2  
 Dynamic data structures, 2-23  
 Dynamic process, 2-2, 2-4  
 Dynamic storage, 2-4

Elements, queue, 1-7  
 End of file, 5-3, 5-6  
 Error, device status code, 4-13  
     to 4-15  
 Error returns, primitive calls,  
     3-5  
 Errors, 4-13 to 4-15, 7-1  
     device handler, 9-18  
     file system, 5-2  
     hardware, 2-11, 7-1, 9-19  
     software, 2-11, 7-1, 9-19  
     system, 2-11, 7-1  
 Exception, 7-1, 9-19  
     declaring, 7-5  
     device handler, 9-19  
     dismissing, 3-31, 7-7  
     dispatching, 7-5, 7-6  
     group, 2-11, 2-14, 3-15, 3-24,  
         3-35, 7-2  
     handler queue, 7-5  
     handlers, 7-7  
     handling, kernel, 2-11  
     mask, 7-7  
     primitives, 3-14  
     procedures, 7-8  
     processes, 7-7  
     routines, 7-8  
     subcode, 7-2  
     type, 2-11, 3-15, 7-1, 7-2  
     wait states, 2-6  
 Exception-processing primitives,  
     1-10  
 Connect to Exception Condition,  
     3-14, 7-7  
 Dismiss Exception Condition,  
     3-31, 7-7  
 Report Exception, 3-70, 7-5  
 Set Exception Routine Address,  
     3-82, 7-5

## INDEX

FALCON. See SBC-11/21  
FALCON-PLUS. See SBC-11/21  
FB.xxx, 5-10  
FIB\$ macro, 5-9  
File identification block (FIB), 5-1, 5-10  
File specifications, 5-6  
File storage, E-16  
File system, 5-1  
    I/O buffers, 5-3, 5-12  
    I/O interface, 5-1  
    QIO subroutine, 5-1  
Floppy disk device handler, 4-43, A-2  
FM\$xxx, 4-7, 4-8, 5-3, 5-4  
FORK\$, 3-42, 8-9, 8-10, 9-17, B-1  
Fork  
    clock, 8-10  
    dispatching, 8-10  
    priority, 9-3, 9-17, B-1  
    processing, 3-42, 8-1, 9-17, B-1  
    routine, 8-9, 9-17  
FSINT\$ macro, 5-8  
FS.xxx, E-20 to E-22  
Full-system mapping, 2-19  
Function code, 4-6, 4-7, 5-4, 6-3  
Function-modifier bit, 4-7, 4-8, 5-3

GELCS, 3-44  
GELMS, 3-46, F-9, F-10  
General-process mapping, 2-18, 2-20  
Get Element, 3-46  
Get Process Status, 3-48  
Get Time of Day and Date, 6-10  
GTD, 6-10  
GTSTS\$, 3-48  
GVALS, 3-50

Hardware priority, 8-1, 9-4, B-1  
Hardware supported, 4-4, 4-5  
HDLC, 4-74, 4-116  
Header, structure, 2-25  
Heap, memory, 2-4  
High segment, memory, 2-16

ID  
    process, 3-8  
    structure, 3-6, 3-7  
IDB, 8-3, 8-6  
    area, 2-33  
Identifier  
    process, 3-8  
    structure, 3-6  
IF\$xxx, 4-7, 5-2 to 5-6

Impure area, kernel, 2-33  
IMPUR\$ macro, 3-52  
Impure segment, memory, 2-16  
Index  
    process ID, 3-8  
    structure, 3-6  
Initialization  
    \$DDINI routine, 9-14  
    device handler, 9-14  
    procedure, 2-3  
    process, 2-2, 2-3  
    structure, 3-7  
Interrupt  
    clock, 6-2  
    dismissing, 8-10  
    dispatch block, 8-3  
        area, 2-33  
    dispatching, 8-3  
        normal, 8-5  
        priority-7, 8-5  
    entry point, 8-3  
    ISR interface, 8-6  
    latency, 8-2  
    level, 8-2, 9-4  
    normal, 8-5  
    priority, 8-1, 9-4  
    priority-7, 8-5  
    service routine, 8-2, 9-16  
        mapping, 2-23  
        Pascal interface, 8-11  
    stack, system, 2-33  
    vector, 8-3  
        setting, 8-6  
Interrupt-management primitives, 1-10  
    Connect to Interrupt, 3-19, 8-1  
    Disconnect from Interrupt, 3-37, 8-2  
I/O, 1-11  
    function, 4-6, 4-7, 5-2  
        priority, 8-1  
        standard, 4-7  
    modifier bit, 4-7, 4-8  
    operation, 1-11  
        reply, 4-12 to 4-16, 5-6  
        request, 4-5 to 4-12  
IODEF include file, H-2, H-14  
IOPKTS include file, 4-3, 4-9, 4-14, 4-15, G-2  
IOT, 3-1  
ISR, 8-2, 9-16  
    mapping, 2-23  
    Pascal interface, 8-11  
    restrictions, 3-20  
ISSxxx, 4-13, 4-14

Kernel, 1-2  
    context switch, 2-12, 2-13  
    exception dispatcher, 2-11, 7-5, 7-6  
    functional components, 1-2

## INDEX

impure area, 2-33  
 interrupt dispatcher, 8-3  
 interrupt exit processing, 8-11  
 mapping, 2-20  
 organization, 1-2  
 pool, 2-20  
 primitives, 1-1, C-1  
     Allocate Packet, 3-12  
     calling from ISRs, 8-9  
     Change Process Priority, 3-17  
     Conditional Get Element, 3-44  
     Conditional Put Element, 3-52  
     Conditional Receive Data, 3-62  
     Conditional Send Data, 3-93  
     Conditional Wait on Queue Semaphore, 3-107  
     Conditionally Allocate Packet, 3-10  
     Conditionally Signal Queue Semaphore, 3-91  
     Conditionally Signal Semaphore, 3-85  
     Conditionally Wait on Semaphore, 3-101  
     Connect to Exception Condition, 3-14, 7-7  
     Connect to Interrupt, 3-19, 8-1  
     Create Process, 3-22  
     Create Structure, 3-26  
     Deallocate Packet, 3-29  
     Define Static Process, 3-33  
     Delete Process, 3-39  
     Delete Structure, 3-48  
     Disconnect from Interrupt, 3-37, 8-2  
     Dismiss Exception, 3-31, 7-7  
     exception processing, 1-10, 7-5 to 7-8  
     Get Element, 3-46  
     Get Process Status, 3-48  
     interrupt-management, 1-10  
     message-transmission plus synchronization, 1-7  
     process management, 1-3  
     process synchronization, 1-5  
     Put Element, 3-56  
     Receive Data, 3-66.1, 4-12  
     Report Exception, 3-70, 7-5  
     Reset Ring Buffer, 3-68  
     resource-management, 1-5  
     Resume Process, 3-72  
     Return Structure Value, 3-58  
     ring buffer, 1-8  
     Schedule Process, 3-76  
     Send Data, 3-77, 4-2  
     Set Exception Routine Address, 3-82, 7-8  
     Signal All Waiters, 3-74  
     Signal Queue Semaphore, 3-87  
     Signal Semaphore, 3-6, 3-89  
     Stop Process, 3-99  
 Suspend Process, 3-97  
     Wait on Queue Semaphore, 3-103, 7-7  
     Wait on Semaphore, 3-105  
 service requests  
     Drop CPU Priority, 3-59  
     Request Fork Processing, 3-42  
 KK device handler, 4-93, H-5  
 KKINC include file, H-2, H-5  
 KK\_read\_data function, H-5, H-6  
 KK\_write\_data function, H-5, H-6  
 KW device handler, 4-49, G-3  
 KWVII-C, 4-49, G-1, G-3  
     NWINC include file, G-2  
 KX device handler, 4-55, 9-1, H-2  
 KXINC include file, H-2, H-3  
 KX/KK communication protocol, 4-55  
     KXLINC include file, H-2, H-24  
     KX\_read\_data function, H-2, H-1  
 KXTII-C peripheral processor, H-1  
 application loading, H-22  
 asynchronous serial line, 4-105, H-14  
 base address range, 4-55, 4-56  
 clock, H-22  
 DMA Transfer Controller (DTC), 4-96, H-7  
 IDs, 4-55, 4-56  
 interface to arbiter, H-5  
 loading, H-22  
 multiprotocol chip, 4-92, 4-105, 4-116  
 parallel ports and timer/counters, 4-120, H-15  
 parallel port DMA, H-28  
 synchronous serial line, 4-116, H-14  
 TMS58, 4-26, 4-92, 9-7, H-7  
 two-port RAM (TPR), 4-55, 4-93, H-2, H-5  
 timer/counters and parallel ports, 4-120, H-15  
 timer restrictions, H-22  
 KXTII-C/arbiter communication protocol, 4-55  
 KXTDFU macro library, 4-100  
 KXT LOAD application load procedure, H-22  
     restrictions, H-23  
 \$KXTQR debug symbol, 4-93  
 \$KXTQW debug symbol, 4-93  
 RX\_write\_data function, H-2, H-3  
  
 LAPB, 4-74, 4-81  
 Latency, 8-2  
 Line-time clock, 6-1  
 Linked lists, 2-32, 2-33  
 List  
     doubly linked, 2-33  
     singly linked, 2-32

## INDEX

- Load application onto EXEII-2  
procedure, H-22
- Low segment, memory, 2-16
- Macro  
CTRCLS, 9-6 to 9-8.2  
DRVCLS, 9-5 to 9-6  
general form  
  prim\$, 3-2  
  prim\$P, 3-2, 3-4, 3-5  
  prim\$S, 3-2 to 3-4  
IMPURS, 3-52  
PDATS, 3-53  
PURE\$, 3-58  
QUEDFS, 2-7, 2-25  
usage rules  
  prim\$, 3-3  
  prim\$P, 3-5  
  prim\$S, 3-4  
xxISZS, 9-9
- MACRO-II  
I/O completion reply, 4-12 to 4-16, 5-6  
I/O request, 4-1 to 4-3, 4-11, 4-12, 5-2  
offset symbols and values, 6-7  
syntax example, document convention, 6-7
- Mapping, 2-16 to 2-23  
  driver, 8-2  
  ISR, 8-2, 9-15  
  type, 2-18 to 2-23  
    device access, 2-19, 2-21  
    Driver Handler, 2-19, 2-22  
    full-system, 2-19  
    general, 2-18, 2-20  
    ISR, 2-23  
    kernel, 2-20  
    privileged, 2-19, 2-22
- Mass Storage Control Protocol (MSCP), 4-37
- Memory mapping, 2-16 to 2-23
- Memory partitioning, 2-16 to 2-23
- Message  
  packet, 2-38, 2-31  
  reply, 4-12 to 4-16  
  request, 4-5 to 4-12, 6-2  
  RTDSP, 5-19
- Message-transmission plus synchronization primitives, 1-3, 1-7
- Conditional Signal Data Semaphore, 3-31
- Conditional Receive Data, 4-62
- Conditional Wait on Queue Semaphore, 3-107
- Conditionally Send Data, 3-23
- Receive Data, 3-66; i  
Send Data, 3-77
- Signal Queue Semaphore, 3-27
- Wait on Queue Semaphore, 3-187
- MIcro/POP-II, 4-37
- SMINST minimum protocol 32015  
  size, 3-24, 4-35, 9-2
- Modem control, 4-65, 4-67, 4-79
- Modem support, 4-67, 4-74
- MSCP disk-class handler, 4-37
- MXVII-A, 4-63
- MXVII-B, 4-63
- Name  
  process, 2-4, 2-5, 3-8  
  structure, 3-6, 3-7
- Normal ISR  
  entering, 8-8  
  executing, 8-9  
  exiting, 8-11
- Null arguments, 3-2
- Offset symbols and values, 6-7  
Optional parameters, document convention, 6-7
- Packet  
  DeAllocate, 3-23  
  message, 2-30, 2-31  
  request, 4-5
- PAB, 2-18 to 2-23, 4-6, 4-8, 8-2, 9-16
- Parallel-line device Handler, 4-54, 4-84
- Parameter. See also Argument  
  null, 3-2  
  optional, document convention, 6-7
- PARSE subroutine, 5-6
- PARSes macro, 5-14
- Partial  
  I/O completion reply, 4-12, 4-16  
  I/O request, 4-12, 4-5 to 4-11
- reserved words, document convention, 6-7
- PCR, 2-2, 2-12 to 2-16
- PDATS, 3-53
- PDB, 3-7  
  ID, 3-8  
  Identifier, 3-8  
  index, 3-9  
  name, 3-8  
  serial number, 3-9  
  usage, 3-7 to 3-9
- POP-II subroutine 4-174  
  convention, D-1
- PELCS, 3-54
- PELMS, 3-56, F-9
- Peripheral processor. See  
  EXEII-2 peripheral processor

**Physical address**, 4-6  
**PIL**, 8-2  
**Pool**  
  free memory, 2-34  
  free packet, 2-34  
  kernel, 2-24  
**Prefix file**, 4-3 to 4-5, 8-2 to  
  9-9  
**Primitive services**, 1-2  
  calls  
    primS variant, 3-2, 3-3  
    primSP variant, 3-2, 3-4, 3-5  
    primSS variant, 3-2 to 3-4  
  error returns, 3-5  
  MACRO-11 interface, 3-1  
  Pascal interface, 3-1  
  requests, 3-1  
**Primitives**, 1-1  
  calling from ISRs, 8-1  
  exception processing, 1-14  
  interrupt management, 1-14  
  message transmission plus  
    synchronization, 1-7  
  process management, 1-3  
  process synchronization, 1-5  
  resource management, 1-3  
  ring buffer management, 1-3  
**Priority**  
  changing process, 3-17  
  device handler, 9-5, 9-6, 9-7  
  interrupt, 9-4  
  process, B-2  
  scheduling hierarchy, B-1  
**Priority-7 ISR**,  
  entering, 8-9  
  executing, 8-9  
  exiting, 8-11  
**Procedure**, 2-2  
  declaration, 2-2  
  exception, 7-8  
  initialization, 2-3, 3-15  
  interrupt, 9-15  
  termination, 9-18  
**Process**, 2-1. (See also **Processes**)  
  blocking, 2-2  
  clock, 6-1  
  codes, state, 2-8  
  components, 2-2  
  concurrent, 2-1  
  context switching, 2-12  
  control block (PCB), 2-2, 2-12  
    to 2-16  
  creating, 2-3  
  definition of, 2-1, 2-2  
  descriptor block (PDB), 3-7 to  
    3-9, 4-5, 4-7  
  define static, 3-3  
  delete, 3-29  
  dynamic, 2-2 to 2-4  
  exception handler, 7-7  
  ID, 3-8  
  identifier, 3-8  
  initialization, 2-1  
  management primitives, 1-5  
    Change Process Priority, 1-17  
    Create Process, 1-22  
    Delete Process, 3-39  
    Get Process Status, 3-12  
    Resume Process, 1-72  
    Synchronize Process, 3-26  
    Stop Process, 3-39  
    Suspend Process, 1-27  
  mapping (702), 2-17, 2-18 to 2-21  
  names, 2-4, 2-6, 3-8  
  priorities, 3-2, 3-4, 3-5, 3-6  
  privileged register, 2-16, 2-20  
    6-1  
  program segmentation, 2-8  
  scheduling, 2-8  
  serial number, 2-2  
  state queue, 2-8  
  states, 2-5 to 2-7  
  status, 2-2, 2-4  
  sub, 2-3  
  suspending, 2-17  
  synchronization primitives, 1-5  
    Conditionally Shared  
      Semaphore, 3-25  
    Conditionally Wait on  
      Semaphore, 3-171  
      Signal All Writers, 3-103  
      Signal Semaphore, 3-20  
      Wait on Semaphore, 3-175  
    unblocking, 2-9  
**Processes**  
  concurrent, 2-1  
  creating, 2-2, 2-3  
  definition of, 2-1, 2-2  
  dynamic, 2-1  
  initializing, 2-3, 2-7  
  static, 2-1  
  system, 1-2, 1-11  
**Process control block (PCB)**, 2-2,  
  2-12 to 2-16  
**Process descriptor block (PDB)**,  
  1-7 to 3-9  
**Program**, 2-1  
  segmentation, 2-16 to 2-22  
**Protocol**, communications line,  
  4-81  
**P-set**, 2-18  
**Pure code**, 2-16, 3-58  
**Pure data**, 2-16, 3-58  
**PURES macro**, 3-58  
**Pit Element**, 3-56  
**P7SYS2**, 3-59  
  
**POL device handler**, 4-96, H-1  
**POL Pascal functions**, H-7  
**JONES macro**, 4-192  
**JDN1 include file**, 4-2, H-8,  
  H-12

## INDEX

QIO subroutine, 5-2  
 QIOS macro, 5-8  
 QIO\$PR, 5-2  
 QUEDE\$ macro, 2-7, 2-25  
 Queue  
     exception, 7-5  
     I/O request, 4-3 to 4-5, 9-15  
     message, 9-15  
     process state, 2-6  
     semaphore, 1-7, 2-27  
     system, 2-31  
  
 Race condition, 2-3  
 RAM, 2-16  
 RBUF\$, 3-69  
 RCVC\$, 3-62  
 RCVD\$, 3-66.1, 4-12, 4-16  
 RD51, 4-37  
 Read\_analog\_continuous procedure,  
     G-8, G-12  
 Read\_analog\_signal procedure,  
     G-8, G-10  
 Read\_analog\_wait procedure, G-8,  
     G-9  
 Read\_counts\_signal procedure,  
     G-3, G-5  
 Read\_counts\_wait procedure, G-3  
 Read-only memory, 2-16  
 Read/write memory, 2-16  
 Real-time clock (KVV11-C), 4-49,  
     G-1, G-3  
 Receive Data, 3-66.1  
 Reentrant code, 2-1, 2-2  
 Reply  
     I/O, 4-12 to 4-16, 5-9  
     semaphore, 4-6 to 4-8  
 Report Exception, 3-70  
 Request message, 4-5 to 4-8, 6-2  
 Request queue names, 4-3 to 4-5  
 Reset Ring Buffer, 3-60  
 Resource-management primitives,  
     1-5  
     Allocate Packet, 3-12  
     Conditionally Allocate Packet,  
         3-10  
     Create Structure, 3-26  
     Deallocate Packet, 3-29  
     Delete Structure, 3-40  
     Return Structure Value, 3-50  
 Resume Process, 3-72  
 Return Structure Value, 3-50  
 REXC\$, 3-70, 7-5  
 RHSDSC include file, G-2  
 RHSLIB object library, G-2, H-2,  
     H-23  
 Ring buffer, 1-8  
     creating, 3-26  
     restrictions, 3-27  
 Ring buffer primitives, 1-8  
     Conditionally Get Element, 3-44  
     Conditionally Put Element, 3-52  
  
 Get Element, 3-46  
 Put Element, 3-56  
 Reset Ring Buffer, 3-60  
 Ring mode, 4-63  
 RL01, 4-31  
 RL02, 4-31  
 RLV11, 4-31  
 RLV12, 4-31  
 RLV21, 4-31  
 ROM, 2-16  
 RSUM\$, 3-72  
 RTDSP, 5-1, 5-4, E-1 to E-20  
 RT-11 media, 5-1, E-1  
 Runtime system, 1-1  
 RX02, 4-43, 5-7  
 RX50, 4-37  
 RXV21, 4-43  
  
 SALL\$, 3-74  
 Save area, 2-12  
 SBC-11/21  
     device handler support, 4-63,  
         4-88  
     exceptions, 7-2, 7-4  
     interrupt levels, 9-4  
     noninterrupt mode PIO support,  
         4-89  
 SCHD\$, 3-76  
 Schedule Process, 3-76  
 Scheduler, 2-12  
 Scheduling hierarchy, B-1  
 SDB, 3-5 to 3-7  
 ID, 3-6, 3-7  
 identifier, 3-6  
 index, 3-6  
 initialization, 3-7  
 name, 3-6, 3-7  
 unnamed, 3-7  
 Segments, program or memory, 2-16  
     to 2-23  
 Semaphore  
     binary, 1-5, 2-26  
     counting, 1-6, 2-27  
     queue, 1-7, 2-27  
 SEND\$, 3-77, 4-2, 4-11  
 Send Data, 3-77  
 Sequence number, 4-6, 4-7, 5-9,  
     6-3  
 SEQ11, D-1  
 SERA\$, 3-82, 7-5  
 Serial number  
     process, 3-8  
     structure, 3-6  
 Service request, 4-2, 6-1  
     Drop CPU Priority, 3-59  
     Fork process, 3-42  
 Set Exception Routine Address,  
     3-82, 7-5  
 Set Time of Day and Date, 6-9  
 SGLCS\$, 3-85  
 SGLQS\$, 3-87

## INDEX

Singly linked lists, 2-32  
 SGNL\$, 3-6, 3-89  
 SGQC\$, 3-91  
 Signal a Semaphore After a Given Time Interval, 6-4  
 Signal a Semaphore Periodically, 6-6  
 Signal All Waiters, 3-74  
 Signal Queue Semaphore, 3-87  
 Signal Semaphore, 3-6, 3-89  
 SNDC\$, 3-93  
 SPND\$, 3-97  
 SSI, 6-4  
 SSP, 6-6  
 Stack, system interrupt, 2-33  
 Stack frame  
     device handler initialization, 9-14  
     exception, 7-8 to 7-10  
     subroutine parameters, D-1 to D-4  
 Standard device handlers, 4-1  
 Standard PDP-11 subroutine calling conventions, D-3  
 Start\_rtclock procedure, G-3, G-7  
 State  
     blocked, 2-5  
     codes, 2-7  
     exception wait active, 2-6  
     exception wait suspended, 2-6  
     process, 2-5 to 2-7  
     ready active, 2-5  
     ready suspended, 2-5  
     run, 2-5  
     transitions, 2-6, 2-7  
     wait active, 2-5  
     wait suspended, 2-6  
 Static process, 2-2, 2-4  
     define, 3-33  
 STD, 6-9  
 Stop Process, 3-99  
 Stop\_rtclock procedure, G-3, G-8  
 STPC\$, 3-99  
 Structure  
     attribute bits, 2-26  
     creating, 3-26  
     deleting, 3-40  
     header, 2-25  
     name, 2-24  
     serial number, 2-26  
     type code, 2-26  
 Structure descriptor block (SDB)  
     ID, 3-6, 3-7  
     identifier, 3-6  
     index, 3-6  
     initialization, 3-7  
     name, 3-6, 3-7  
     unnamed, 3-7  
     usage, 3-5, 3-6  
 Subroutine calling conventions  
     normal calling conventions, D-1  
     standard PDP-11 (SEQ11) calling conventions, D-3  
 Suspend Process, 3-97  
 Suspension, process, 2-10  
 Symbols, offset values, x9  
 System processes, 1-2, 1-11  
     clock process, 1-12, 6-1  
     common area, 2-20, 2-33, 2-34  
     standard device handlers, 1-11, 4-1  
 System queues, 2-31 to 2-33  
  
 Tape drive, TU58, 4-26, 4-92, E-1  
 Terminal I/O. See XL device handler  
 Ticks, 6-1  
 Timer services, 6-1  
 Traps, 7-1, 9-19  
  
 Unblocking, process, 2-9  
 Units  
     device controller, 4-3, 4-4  
     number, 4-6, 4-7, 9-5, 9-6, 9-9  
 Unnamed structures, 3-7  
  
 Vectors, 8-3  
     setting, 8-6  
  
 WAIC\$, 3-101  
 WAIQ\$, 3-103, 7-7  
 WAIT\$, 3-105  
 Wait-active state, 2-5  
 Wait on Queue Semaphore, 3-103, 7-7  
 Wait on Semaphore, 3-105  
 Wait-suspended state, 2-6  
 WAQC\$, 3-107  
 Write\_analog\_wait procedure, G-13  
 Writing device handlers, 9-1 to 9-20  
  
 X.25 communications protocol, 4-74, 4-116  
 XA device handler, 4-59  
 XL device handler, 4-63, A-19, F-9  
     for KXT11-C, 4-105, H-14  
 XP device handler, 4-74  
 XS device handler, 4-116, H-14  
 xxDRV.MAC or xxDRV.PAS, 9-10  
 xxISZ\$, 9-9  
 xxPFX.MAC or xxPFX.PAS, 9-3

INDEX

YA device handler, 4-84, A-43  
YF device handler, 4-88  
YFDRV1 include file, 4-89  
YFDRV1 source file, 4-89  
YK\_CLEAR\_TIMER function, H-15,  
  H-22  
YK device handler, 4-120, H-15  
YKINC include file, H-2, H-15,  
  H-16  
YK\_PORT\_READ function, H-15,  
  H-16, H-18

YK\_PORT\_WRITE function, H-15,  
  H-17, H-18  
YK\_READ\_TIMER function, H-15,  
  H-21  
YK\_SET\_PATTERN function, H-15,  
  H-18  
YK\_SET\_TIMER function, H-15, H-20  
  
Zx device identifiers, 9-2

MicroPower Pascal  
Runtime Services Manual  
AD-M391B-T1

**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 \_\_\_\_\_ Telephone \_\_\_\_\_

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

**SSG/ML PUBLICATIONS, MLO5-5/E45  
DIGITAL EQUIPMENT CORPORATION  
146 MAIN STREET  
MAYNARD, MA 01754**

— — Do Not Tear — Fold Here — — — — —

Cut Along Dotted Line