MicroPower/Pascal Run-Time Services Manual

Order No. AA–M391D–TK

digital

software

MicroPower/Pascal Run-Time Services Manual

Order No. AA–M391D–TK

June 1987

This manual contains the run-time services information required for designing and developing MicroPower/Pascal microcomputer application programs. Run-time services include kernel processing of primitive requests, interrupts, exceptions, and clock services.

This manual also describes the configuration file macros required for building a target memory image.

This manual supersedes the *MicroPower/Pascal Run-Time Services Manual*, AA–M391C–TK.

Operating System and Version:	Micro/RSX Version 3.0 RSX–11M Version 4.2 RSX–11M–PLUS Version 3.0 RT–11 Version 5.2 VAX/VMS Version 4.0
Software Version:	MicroPower/Pascal–Micro/RSX Version 2.4 MicroPower/Pascal–RSX Version 2.4 MicroPower/Pascal–RT Version 2.4 MicroPower/Pascal–VMS Version 2.4

Digital Equipment Corporation   Maynard, Massachusetts

First Printing, February 1982
Updated, June 1982
Updated, October 1982
Updated, February 1983
Revised, July 1983
Updated, February 1984
Revised, June 1985
Updated, April 1986
Updated, October 1986
Revised, June 1987

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

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

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

Copyright © 1982,1983,1984,1985,1986,1987 by Digital Equipment Corporation

All Rights Reserved.

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

The following are trademarks of Digital Equipment Corporation:

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

ML-S692

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

Contents

Preface

Chapter 1 Introduction

Chapter 2 Processes and System Data Structures

Chapter 3 MACRO–11 Primitive Service Requests

Chapter 4 System Configuration Macros

Chapter 5 Dynamic RAM Allocation and Region Sharing

Chapter 6 Exception Processing

Chapter 7 Interrupt Dispatching and Interrupt Service Routines

Appendix A Scheduling Hierarchy and Recommended Process Priorities

Appendix B MACRO–11 Subroutine Calling Conventions

Index

Examples

Figures

Tables

Preface

This manual contains the run-time services information required for designing and developing MicroPower/Pascal microcomputer application programs. Run-time services include kernel processing of primitive requests, interrupts, exceptions, and clock services.

This manual also describes the configuration file macros required for building a target memory image.

Structure of This Document

Seven chapters and two appendixes make up this manual, as follows:

- Chapter 1 presents an overview of the MicroPower/Pascal run-time 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 this information before attempting to write application code using the run-time services described in the remainder of this manual.
- Chapter 3 gives detailed descriptions for each of the MACRO–11 primitive service requests. Primitive services provided by the MicroPower/Pascal kernel are also accessible to applications written in Pascal. See the *MicroPower/Pascal Language Guide* for details on issuing kernel primitive requests in Pascal programs.
- Chapter 4 provides complete user information for the configuration macros required for building a kernel and memory image file. Information is provided for both Pascal and MACRO–11 users.
- Chapter 5 describes the dynamic-RAM allocation and mapping services that are available to both the Pascal and MACRO–11 user.
- Chapter 6 describes MicroPower/Pascal exception processing. (Exceptions are hardware or software errors or traps that may occur when application programs are executed or debugged on the target system in the real-time environment.)
- Chapter 7 describes kernel interrupt dispatching, interrupt service routines (ISRs), and fork routines.

- Appendix A describes the scheduling-priority hierarchy throughout a MicroPower/Pascal application.
- Appendix B explains how to interface subroutines written in MACRO–11 with Pascal programs.

Intended Audience

The content of this manual is based on the assumption that you are familiar with either Pascal or MACRO–11. All MicroPower/Pascal microcomputer software development is done with one or both of those development languages. Additional run-time services reference information for writing applications in Pascal is contained in the *MicroPower/Pascal Language Guide*.

Conventions Used in This Document

The following conventions are used in this document:

- Pascal-reserved words that must not be abbreviated are shown in uppercase characters in syntax examples. Within those examples, lowercase characters are used for variable parameters (or other syntax elements) that you may choose for your application.
- 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 Section 3.1.1, which describes the general form and usage rules for the prim$ macro variant.
- In this manual, some MACRO–11 syntax examples are shown with long macro invocations continued on a second line—for example, the CRP$ and DFSPC$ macro calls. However, when writing source code in MACRO–11, you must keep each macro invocation on a single line.
- 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.

Associated Documents

The following software documentation is required for complete reference purposes:

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

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

- Microcomputer handbooks
  - *Microcomputers and Memories*
  - *Microcomputer Interfaces Handbook*
- *M8063 Falcon SBC–11/21 Single-Board Computer User's Guide* (required when developing SBC–11/21 applications)

- *SBC-11/21-PLUS Single-Board Computer User's Guide* (required when developing SBC-11/21-PLUS applications)
- *KXJ11-CA Single-Board Computer User's Guide* (required when developing KXJ11-CA applications)
- *KXT11-CA Single-Board Computer User's Guide* (required when developing KXJ11-CA applications)
- *DPV11 Serial Synchronous Interface Technical Manual* (required when developing applications that use DPV11 communications hardware)
- *LSI-11 Analog System User's Guide* (required when developing applications that use the ADV11-C, AAV11-C, AXV11-C, or KWV11-C analog I/O boards)
- *MSCP Basic Disk Functions Manual* (required when developing applications that use MSCP disk-class devices)
- Additional hardware documentation for microcomputer hardware presently not covered in the microcomputer handbooks

Chapter 1
Introduction

This manual describes the organization of the MicroPower/Pascal run-time system and the services that the MicroPower/Pascal kernel provides for user programs. The explicit, user-requested services provided by the kernel are the real-time primitive operations, described in Chapter 3. (Standard device-I/O services, file system services, and communications support are provided by system processes and are described in the *MicroPower/Pascal I/O Services Manual.*) Implicit services provided by the kernel include process scheduling (Chapter 2), trap/exception processing (Chapter 6), and interrupt dispatching (Chapter 7).

Chapter 2 provides an overview of the process/kernel relationship. The chapter discusses the dynamic characteristics of a MicroPower/Pascal concurrent process and gives a detailed description of process states, scheduling, and the effects of process-mapping type in a mapped environment. Chapter 2 also describes the system data structures the kernel uses to implement primitive operations. Chapter 4 describes the system configuration macros used at build time to determine the application's run-time environment. Other chapters provide supporting information.

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.

However, some of the information (particularly Chapter 3, which describes MACRO–11 primitive service requests) is pertinent only to MACRO–11 programmers. Analogous information for Pascal programmers is provided primarily in Part II of the *MicroPower/Pascal Language Guide.*

1.1 The MicroPower/Pascal Run-Time System

The MicroPower/Pascal run-time system is the collection of DIGITAL-supplied software that resides in the target system and provides the execution-time environment for application programs. The run-time system consists of the MicroPower/Pascal kernel, numerous system-level processes, and, optionally, a resident, shared library.

The kernel provides the set of basic operations, called primitives, that are required for concurrent programming. These primitives implement, for example, process creation and deletion, process synchronization, and interprocess communication. The system processes basically provide I/O support. Optionally, you can build a run-time shared library containing common code that would otherwise be duplicated in several or many processes' physical address space. Typically, the library would contain Pascal OTS routines but could be used for processes implemented in MACRO–11 as well. Such a library may be used to achieve optimal memory utilization in some target environments.

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, which 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.

The DIGITAL-supplied system processes provide device-handling services for commonly used I/O devices and device interfaces, file system support, and network or local communications support. Application processes written in MACRO–11 obtain these services by using queue semaphore primitives to send request messages to the appropriate system process. Pascal-implemented processes normally obtain the same services through various Pascal I/O statements. The system processes are included in the target system during system building on an individual, as-needed basis.

1.2 Kernel Organization

The MicroPower/Pascal kernel consists of many small program modules with well-defined functions and interfaces. The highly structured character of the kernel not only makes it 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:

- The scheduler, which allocates the CPU to processes, according to priority, on an event-driven, preemptive basis
- The primitive service routines—the many modules that implement the individual primitive operations requested by processes
- The primitive dispatcher, which receives all primitive service requests and passes control to the appropriate primitive service module
- The interrupt dispatcher, which receives all device interrupts and passes control to appropriate service routines, providing the necessary entry and exit processing

- The trap handler/exception dispatcher, which receives all exception conditions—actual and simulated processor traps—and transfers control as required for handling the exception
- The system-initialization routine, which initializes kernel data structures and installs static processes at start-up/restart time

The primitive service modules constitute by far 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 approximately 60 primitive operations for concurrent programming. Most of those primitives can be grouped into 10 major categories, as follows:

- Process management—Creation, deletion, suspension, resumption, and forced termination of processes.
- Resource management—Creation and deletion of data structures, such as semaphores and ring buffers, and allocation and deallocation of message packets.
- Process synchronization—Synchronization of cooperating processes by means of Signal and Wait operations on binary and counting semaphores.
- Message transmission and synchronization—Interprocess communication through operations on queue semaphores and combinations of packet queuing/dequeuing and Signal and Wait operations.
- Ring buffer management—Variable-length data transfers between processes, through ring buffers, without the need for close synchronization between putters and getters.
- Exception management—Control of hardware and software exception-condition dispatching to an appropriate exception-handling process or exception service routine and reporting of a software exception by a process. (The hardware-detected events reported by processor traps other than IOT or power-fail constitute the MicroPower/Pascal hardware exceptions.)
- Interrupt management—Control of interrupt dispatching; used only by processes that manage an I/O device.
- Timer services—Control of system time and process "sleeping" for a desired time.
- Dynamic region allocation and management (DRAM)—Acquisition of unused regions of memory, sharing of a region of memory with another process, and the mapping operations connected with the use of those regions.
- Logical-name services—Creation, translation, and deletion of logical names.

Primitives are described briefly in the following subsections. Chapter 3 contains complete descriptions for the MACRO–11 programmer. See Part II of the *MicroPower/Pascal Language Guide* for a description of the Pascal primitive service request interface. (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 between MACRO and Pascal usage are indicated in the next subsection.)

Note

Several assembly-time macros—Define Static Process (DFSPC$), Define a Pure Program Instruction Section (PURE$), Define a Pure Program Data Section (PDAT$), and Define an Impure Program Data Section (IMPUR$)—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$) and Enter Normal ISR State (P7SYS$).

1.2.1.1 Process-Management Primitives

This category of primitives contains the following:

- Create Process—Lets an existing process create a new process dynamically 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: CRPC$
    - Pascal equivalent: Process invocation statement
- Delete Process—Lets a process delete itself from the system; the only valid 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–11 service request name: DLPC$
    - Pascal equivalent: None
- Suspend Process—Lets a process 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–11 service request name: SPND$
    - Pascal equivalent: SUSPEND function
- Resume Process—Lets a process reactivate another suspended process.
    - MACRO–11 service request name: RSUM$
    - Pascal equivalent: RESUME function
- Stop Process—Lets one process force another process or itself to execute its termination routine or (Pascal) TERMINATE procedure. (The "stopped" process must delete itself to go away.)
    - MACRO–11 service request name: STPC$
    - Pascal equivalent: STOP procedure
- Get Process Status—Lets one process obtain information about the status of either itself or another process.
    - MACRO–11 service request name: GTST$
    - Pascal equivalent: GET_STATE procedure

- Change Process Priority—Lets a process modify its own or another process's scheduling priority. Normally, this primitive is used to lower priority from a very high start-up value used only for initialization code. (In Pascal, the INITIALIZE procedure attribute indirectly serves this purpose.)
  - MACRO-11 service request name: CHGP$
  - Pascal equivalent: CHANGE_PRIORITY procedure
- Schedule Process—Lets a process relinquish control of the CPU to another process of equal priority, if one is ready to execute.
  - MACRO-11 service request name: SCHD$
  - Pascal equivalent: SCHEDULE procedure
- Define Stop Flag Address—Lets a process defer the effect of a Stop Process request issued by another process.
  - MACRO-11 service request name: SSFA$
  - Pascal equivalent: DEFINE_STOP_FLAG procedure

1.2.1.2 Resource-Management Primitives

This category of primitives contains the following:

- Create Structure—Creates a system data structure (a semaphore, ring buffer, or unformatted structure) in kernel data space.
  - MACRO-11 service request name: CRST$
  - Pascal equivalents: { CREATE_BINARY_SEMAPHORE function
CREATE_COUNTING_SEMAPHORE function
CREATE_QUEUE_SEMAPHORE function
CREATE_RING_BUFFER function }
- Delete Structure—Deletes a system data structure.
  - MACRO-11 service request name: DLST$
  - Pascal equivalent: DESTROY procedure
- Get Structure Value—Obtains the characteristics (for example, type) and value of a system data structure.
  - MACRO-11 service request name: GVAL$
  - Pascal equivalent: GET_VALUE procedure
- Allocate Packet—Obtains an empty message packet from the kernel's free-packet pool (returns a pointer).
  - MACRO-11 service request name: ALPK$
  - Pascal equivalent: ALLOCATE_PACKET procedure

- 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-11 service request name: ALPC$
  - Pascal equivalent: COND_ALLOCATE_PACKET function
- Deallocate Packet—Returns a message packet to the kernel's free-packet pool, thus freeing the packet for reuse.
  - MACRO-11 service request name: DAPK$
  - Pascal equivalent: DEALLOCATE_PACKET procedure

1.2.1.3 Process-Synchronization Primitives

The primitives in this category operate on a binary or 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 (B) semaphore are:

```
SIGNAL(B):  If B = 0 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
```

A Signal of a binary semaphore having a value of 0 allows 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 proceeds 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 allows one subsequent Wait to proceed without blocking the process. Unlike binary semaphores, however, successive Signals without intervening Wait operations are not lost. Each Signal is counted and allows one Wait to proceed without blocking.

The process-synchronization primitives are:

- Signal Semaphore—Performs an unconditional Signal operation on a specified binary or counting semaphore.
  - MACRO-11 service request name: SGNL$
  - Pascal equivalent: SIGNAL procedure

- Wait on Semaphore—Performs an unconditional Wait operation on a specified binary or counting semaphore.
  - MACRO–11 service request name: WAIT$
  - Pascal equivalent: WAIT procedure
- Conditionally Signal Semaphore—Performs a conditional Signal operation, which increments the binary or counting 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: SGLC$
  - Pascal equivalent: COND_SIGNAL function
- Conditionally Wait on Semaphore—Performs a conditional Wait operation, which decrements the binary or counting 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: WAIC$
  - Pascal equivalent: COND_WAIT function
- Wait on Any Semaphore—Performs either a conditional or unconditional Wait operation on up to four specified binary or counting semaphores, with an optional timeout if a Signal does not occur within a given time.
  - MACRO–11 service request name: WAIA$
  - Pascal equivalent: WAIT_ANY procedure
- Signal All Waiting Processes—Performs a special form of Signal operation, which unblocks any and all processes that may be waiting on the specified binary or counting semaphore and sets the semaphore value to 0 unconditionally.
  - MACRO–11 service request name: SALL$
  - Pascal equivalent: SIGNAL_ALL procedure

1.2.1.4 Message-Transmission Plus Synchronization Primitives

The primitives in this category operate on queue semaphores and combine message-packet transmission and reception 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 message 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, usually packets, 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:

- Signal Queue Semaphore (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: SGLQ$
  - Pascal equivalent: PUT_PACKET procedure
- Wait on Queue Semaphore (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-11 service request name: WAIQ$
  - Pascal equivalent: GET_PACKET procedure
- 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: SGQC$
  - Pascal equivalent: COND_PUT_PACKET function
- Conditionally Wait on Queue Semaphore—Performs a conditional Wait on Queue operation, which removes a packet pointer from the semaphore's queue and returns it to the requester only if a packet is on the queue (that is, if 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: WAQC$
  - Pascal equivalent: COND_GET_PACKET function
- Wait on Any Queue Semaphore (Get Packet Any)—Performs either a conditional or unconditional Wait operation on up to four specified queue semaphores, with an optional timeout if a packet does not arrive within a given time.
  - MACRO-11 service request name: WAQA$
  - Pascal equivalent: GET_PACKET_ANY procedure

- Send Data by 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
- Receive Data by 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 blocks if a packet is not immediately available.
  - MACRO–11 service request name: RCVD$
  - Pascal equivalent: RECEIVE procedure
  - Pascal variant: RECEIVE_ACK procedure
- 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 through 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
- 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: RCVC$
  - Pascal equivalent: COND_RECEIVE procedure
  - Pascal variant: COND_RECEIVE_ACK procedure
- Receive Data Through Any Queue Semaphore—Performs a complex Receive operation on up to four specified queue semaphores, then copies data from the packet obtained from any one of those queues into a caller-specified data area, and finally deallocates the packet (that is, returns the packet to the free-packet pool). The calling process may or may not block if a packet is not immediately available, depending on the form of the call, and the Wait can optionally be timed out if a packet is not sent within a given time.
  - MACRO–11 service request name: RCVA$
  - Pascal equivalent: RECEIVE_ANY procedure
  - Pascal variant: RECEIVE_ACK_ANY procedure

In a mapped environment, a process must have privileged or driver mapping to 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 to access (write into or read from) the packet.

1.2.1.5 Ring Buffer Primitives

The primitives in this category operate on ring buffer structures, which facilitate variable-length data transfers, normally of character or byte-oriented data, between processes, without the need for tight, signal/wait synchronization between them. The size, or capacity, of a ring buffer is determined when the structure is created; the size can be from 8 bytes to just less than 8K bytes. The ring buffer primitives are:

- Get Element—Moves a specified number of bytes of data from a ring buffer to a data area specified by the requester. If the buffer does not have 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: GELM$
  - Pascal equivalent: GET_ELEMENT procedure
- Put Element—Moves a specified number of bytes of data from a data area specified by the requester to the ring buffer. If the buffer has insufficient space to accommodate the new element, the calling process blocks until sufficient space becomes available because of subsequent Get operations.
  - MACRO–11 service request name: PELM$
  - Pascal equivalent: PUT_ELEMENT procedure
- Conditionally Get Element—Obtains 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 have 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: GELC$
  - Pascal equivalent: COND_GET_ELEMENT function
- Conditionally Put Element—Places 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: PELC$
  - Pascal equivalent: COND_PUT_ELEMENT function

- Get Element Any—Moves a specified number of bytes of data from any one of up to four specified ring buffers to a data area specified by the requester. If the buffer does not have enough data to satisfy the request, the calling process may or may not block, depending on the form of the call; optionally, the wait for data can be timed out after a given time.
  - MACRO–11 service request name: GELA$
  - Pascal equivalent: GET_ELEMENT_ANY procedure
- Reset Ring Buffer—Empties a specified ring buffer of all data.
  - MACRO–11 service request name: RBUF$
  - Pascal equivalent: RESET_RING_BUFFER procedure

1.2.1.6 Exception-Processing Primitives

This category contains the following primitives:

- Connect to Exception Condition—Lets a process establish itself as an exception handler for processes that belong to a given exception-handling group.
  - MACRO–11 service request name: CCND$
  - Pascal equivalents: $\left\{\begin{array}{l}\text{CONNECT\_EXCEPTION procedure}\\ \text{DISCONNECT\_EXCEPTION procedure}\end{array}\right\}$
- Dismiss Exception Condition—Lets an exception-handler process dismiss an exception, releasing the faulting process from exception-wait state for further disposition by the kernel.
  - MACRO–11 service request name: DEXC$
  - Pascal equivalent: RELEASE_EXCEPTION procedure
- Set Exception Routine Address—Lets any process 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: SERA$
  - Pascal equivalents: $\left\{\begin{array}{l}\text{ESTABLISH procedure}\\ \text{REVERT procedure}\end{array}\right\}$
- Report Exception—Lets a process report a software exception condition or force a hardware exception (simulate a processor trap).
  - MACRO–11 service request name: REXC$
  - Pascal equivalent: REPORT procedure

1.2.1.7 Interrupt-Management Primitives

The two primitive operations in this category involve interrupt service routines (ISRs):

- Connect to Interrupt—Lets a device-handling, or driver, process connect an ISR to a specified interrupt vector. (A Pascal variant of this primitive lets a process connect a binary or counting semaphore to an interrupt vector indirectly.)
  - MACRO–11 service request name: CINT$
  - Pascal equivalent: CONNECT_INTERRUPT procedure
  - Pascal variant: CONNECT_SEMAPHORE procedure
- Disconnect from Interrupt—Lets a driver process disconnect an ISR from a specified interrupt vector.
  - MACRO–11 service request name: DINT$
  - Pascal equivalent: DISCONNECT_INTERRUPT procedure
  - Pascal variant: DISCONNECT_SEMAPHORE procedure

1.3 Overview of System Processes

System processes provide commonly used hardware-oriented services for user programs. These processes include many standard (DIGITAL-supplied) device drivers; the ancillary control process (ACP), which provides RT–11-compatible file management and/or non-file-structured device access; and several network and point-to-point communications support processes. (Two driver processes are supplied specifically for communication between a Q-bus arbiter processor and one or more KXT11–CA or KXJ11–CA IOP slave processors.) The *MicroPower/Pascal I/O Services Manual* describes those system processes in detail.

A device driver is a process, or a family of cooperating processes, that accepts requests for device-level I/O operations from other processes. Device drivers communicate and synchronize with other processes in the application through standard primitive operations. I/O service requests for a particular hardware device are passed to the device driver in the form of a request message (queue packet). Each driver maintains a request queue semaphore through which device-level I/O requests are passed. After receiving a request, the driver performs all process-level, interrupt-level, and fork-level processing for the requesting process. When the I/O operation has been completed, the driver signals the requesting process and returns completion status by means of a reply message packet. The reply message packet indicates successful completion or error and other information, such as number of bytes successfully transferred, as applicable.

Standard I/O functions generally supported by device drivers 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.

Device drivers can be written in either MACRO–11 or Pascal, with some restrictions on Pascal implementation, and driver processes can be accessed by other processes written in either Pascal or MACRO–11. All standard DIGITAL-supplied device drivers are written in MACRO–11 for maximum efficiency and flexibility.

The ACP and, optionally, the network service process (NSP) provide a higher level of control that is "layered" on top of the driver-level processes, eliminating the need for user processes to talk to drivers directly. Access to drivers, the ACP, and the NSP by a user process implemented in Pascal is generally transparent, obtained through OPEN and other MicroPower/Pascal I/O statements.

1.4 Resident Shared Libraries

A resident shared library, or run-time library, allows two or more static processes to share "library" code at run time that would otherwise have to be merged into each process's object code at build time. Such libraries permit a savings in physical memory requirements by eliminating duplication of pure code across static processes. Resident shared libraries are possible in all target hardware environments, but the cost/benefit tradeoffs vary with the environment. For an unmapped target system, use of a resident shared library is a clear win if the application contains more than one user static process. (In unmapped applications, all user processes are often part of one static process family for the most economical implementation.) An unmapped environment has no virtual address-space considerations, and, if the application is intended for ROM, use of a shared library can make PROM burning less laborious.

For a mapped target system with supervisor mode, such as an LSI–11/73-based target, use of a shared library is also a clear win, since such a library has a separate supervisor-mode mapping and thus does not impinge in a negative way on the virtual address space of a user process that references the library. Again, if the application is intended for ROM, use of a shared library can simplify PROM burning.

For a mapped target system without supervisor mode, such as an LSI–11/23-based target, the tradeoff considerations are somewhat complex, because the entire shared library is mapped into the virtual address space of any referencing process. The shared library will contain all the code that any referencing process uses and thus may contain much code that a given process does not need. That is to say, a shared library may "steal" a significant amount of virtual address space from user static processes, due both to unused code and to PAR boundary alignment problems. Therefore, if a given static process family (that is, an individual build unit) is approaching the limits of its virtual address space, the tradeoff of increased virtual address space for the process in question against decreased physical memory for the entire application may not be possible without redesign of the user static processes. (An application could contain multiple shared libraries, but that option complicates the application-building procedure considerably, since the "automatic" MPBUILD facility cannot readily be used to achieve it.) Again, if the application is intended for ROM, use of a shared library can simplify PROM burning.

The choice of whether to use an object-time (nonshared) library or a resident shared library is made at application build time. An application may consist of a mixture of static processes that do and do not use the resident shared library code. In a mapped system, a member process of a static process family that does not reference a shared library is not affected by the existence of that library. See Chapter 2 for a description of shared supervisor-mode library mapping and the MicroPower/Pascal system user's guide for your host system for details of building an application with a shared library.

Chapter 2

Processes and System Data Structures

This chapter begins with a general description of processes and then presents 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 an 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. That 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. However, the MicroPower/Pascal process-synchronization primitives 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 a number of autonomous subprograms that are scheduled independently when triggered by appropriate events. Such 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 interrupts. Thus, the process approach 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 several subareas, or windows. Each window is to be a virtual display that is updated independently in response to a 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 lets the programmer manage each window with a separate process and 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, or PCB), created and maintained by the kernel, that reflects the context of the process at any given point. The PCB 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. (In a dedicated, real-time environment, this instruction sequence is often nonterminating except under special conditions.) 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, such as the process stack and any static variables, that are unique to the process, plus any shared data.

An instruction sequence, if reentrant, may be shared (concurrently executed) by several processes. Thus, a process represents one specific invocation of an instruction sequence as an independent scheduling unit. The PCB 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. The kernel's initialization (INIT) routine creates a PCB 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 running 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 and heap space allocated to the main program, constitute the data segments associated with the static process. (The heap is used dynamically for NEW and DISPOSE and for the stack and local variables of any dynamic processes created by the static process.)

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 program, the procedure is executed before the corresponding static-process code (main program body) is initially executed. (No procedure call is required.)

Furthermore, the initialization procedure has a default scheduling priority of 248, the highest recommended start-up priority value for a user process. The static process itself is scheduled at the running priority specified or defaulted to in the program heading. (Running priorities in general should not exceed 247 and should be less than 160 for normal user processes. See Appendix A for recommended process priorities.) The combination of implicit precedence of execution and special start-up priority guarantees that the initialization procedure will run not only before its associated static process but also before any other Pascal static process begins execution—assuming that the initialization code does nothing that might cause it to block, which it should not do.

The purpose of the initialization procedure is to permit creation of any system data structures—semaphores, ring buffers, or shared regions, for example—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, thereby avoiding the potential race condition that could arise if one process were to depend on another to start first. (Relative running priorities should not be relied on to ensure the order in which processes start up and are not intended for that purpose.)

In MACRO–11, a static process is defined by the Define Static Process (DFSPC$) assembly-time macro; see Section 3.15. This macro produces a block of information used by the memory image builder (MIB) utility and the kernel's INIT routine. The information includes the initial address of the instruction sequence to be executed, the size and location of the process stack, the run-time 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 248 or higher, as specified in the DFSPC$ macro, in order to execute its initialization code. Immediately after the initialization processing, the process uses the Change Priority (CHGP$) primitive to drop its priority to the desired operating level; the process can then enter its main code, corresponding to the Pascal main program body. (The CHGP$ primitive call always implies a scheduling operation.) This strategy is in fact the same as that used by the Pascal OTS to implement Pascal initialization procedures, of which a program may have several.

A dynamic process is created by the action of another process during system execution. The action consists of a request to the kernel's process-creation service, which creates a process control block (PCB) 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 the static process in the sense that the instruction and data segments of the created process must be located within the address space of the static process (that is, within the same object program). 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 syntactically similar to a procedure call, a process invocation 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 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 from the heap for each dynamic process, as well as a separate stack.

In MACRO-11, a dynamic process is created by a Create Process (CRPC$) service request; see Section 3.10. The request specifies the initial address of the instruction sequence to be executed, the stack address, run-time 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—which is the only valid way for a process to terminate, assuming that such 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 must indeed do so).

The MicroPower/Pascal compiler and object-time system (OTS) does, however, impose its own default structure on a process family with respect to the longevity of processes and process-local variables. Essentially, the compiler and OTS provides a method for proper sequencing of process termination, as explained below, in order to safeguard data that is shared between processes. Since the compiler applies the same scoping rules to PROCESS declarations as to PROCEDURE declarations, it can control the scope of variables declared in and 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 process's memory stack) when the process is created, unless the variables 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.

The MicroPower/Pascal method for sequencing the termination of Pascal-implemented processes causes a process to wait for the termination of all processes created by it before it will terminate. That is the default condition for all process invocations and assumes that the created, or child, process has a data dependency on the parent process. The RELATIONSHIP parameter of the process-invocation statement lets you modify the default termination condition for the creating, or parent, process. The lifetime of data items used by but not declared within the child process (variables local to the parent, passed parameters, or new variables generated by the parent) must be considered when you determine the correct setting of the RELATIONSHIP parameter: DEPENDENT or INDEPENDENT.

For a dynamic process to be safely declared INDEPENDENT of its parent, all data items used by the former must continue to exist for the lifetime of the created process. (MicroPower/Pascal guarantees that PROGRAM-level variables and variables declared with the AT, EXTERNAL, GLOBAL, or STATIC attributes exist for at least as long as any process in the static process family exists.) For a process declared as DEPENDENT, the MicroPower/Pascal compiler and OTS make sure that the creating process is never deleted (does not actually terminate) before any of its dependent-child processes terminate, although it may have stopped executing. That is, the storage for a given process is not released, and the process is not deleted, until all dependent processes terminate, even though the process has logically terminated either by "reaching" its END statement" or by executing its termination procedure.

When a process terminates, the local variables (VAR declarations) and any non-VAR formal parameters cease to exist. Therefore, a created process that uses those kinds of data items belonging to the parent process is necessarily DEPENDENT on the parent. Such uses can occur in four ways:

- Up-level addressing. This occurs when a process is declared within the body of another process. Since the typical and proper use of this type of nesting is to take advantage of up-level addressing, the created process can always be said to be DEPENDENT on the creating process.
- VAR formal parameters. If the created process accepts a VAR formal parameter and the creating process passes, as the corresponding actual, one of its local variables or one of its non-VAR formal parameters, the created process is DEPENDENT on the creating process.
- Pointer-type formal parameters. If the created process accepts a pointer to a data item and the creating process passes, as the corresponding actual, a pointer to one of its local variables or to one of its non-VAR formal parameters, the created process is DEPENDENT on the creating process.
- Records containing pointers. If the created process uses a record that contains a pointer to one of the creating processes' local variables or one of the creating processes' non-VAR formal parameters, the created process is DEPENDENT on the creating process. (The manner in which the created process gains access to the record does not affect the validity of this rule.)

If any of the four conditions is met, the created process should be invoked with the default RELATIONSHIP:=DEPENDENT parameter, which will direct MicroPower/Pascal to sequence the termination of the respective processes. If the stated conditions indicate that the created process is in fact dependent but the RELATIONSHIP:=INDEPENDENT parameter is used, you must make sure that the data item in question continues to exist. Otherwise, unpredictable results may occur.

When determining the relationship of processes, you should examine only the two directly related processes: creating and created. That is, if the creating process was itself created by another process, their parent/child relationship need not be considered.

Three programming errors are commonly associated with the passing and/or sharing of data items between processes:

- Passing a pointer to a record obtained by means of NEW to another process and subsequently disposing of the record before the sharing process is finished with it.
- Declaring a process within the body of a PROCEDURE or FUNCTION. As when a process terminates, the local variables and the non-VAR formal parameters cease to exist when a PROCEDURE or FUNCTION exits. The process-termination sequencing method in no way guarantees that PROCEDURE or FUNCTION local variables survive the created process.
- Concurrent use of variables between processes without use of a mutual-exclusion (mutex) mechanism, such as a semaphore or mutex.

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 run-time name in the program that defines the process. A run-time 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 identifies 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.

Process names must be unique among not only all named processes throughout the system but also all named system structures. That is, a process name must not duplicate the name of any coexisting semaphore, ring buffer, or other type of dynamic data structure. Violation of this rule will cause errors during execution. (The names of system structures created by DIGITAL-supplied system processes, such as device drivers, always contain a dollar sign ($) character. You should therefore avoid that character in all user-specified names.)

Since run-time names are fixed-length character strings, both case and trailing blanks are significant. Thus, the name 'abc123' is not equivalent to 'ABC123', and 'ABCD ' is not equivalent to 'ABCD'.

In Pascal, a static process gets its run-time name from the compile-time program name specified in the program heading; the name is either truncated to six characters or padded with trailing spaces to that length, as necessary. A dynamic process gets its run-time 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 run-time 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 run-time name is specified directly in the Define Static Process (DFSPC$) macro call and indirectly in the Create Process (CRPC$) service request. Section 3.1.6 discusses the process descriptor block.

Every process is in one and only one state at any time. The kernel supports the following eight process states:

1. Run: the state of the process eligible for execution. This process may be executing at process level, may be executing a primitive operation in the kernel, or may be interrupted. (An interrupt does not of itself 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, suspends, or deletes itself; is preempted by a higher-priority process becoming ready to execute; or causes an exception.

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 to the processor whenever the running process relinquishes control or is preemptable.

3. Wait active (blocked): the state of a process forced to wait (defer execution) until a particular event occurs or a given resource becomes available. A waiting process is always blocked on a blocking structure (for example, semaphore or a ring buffer). When unblocked, the process changes to the ready-active state. See Section 2.1.4.2.

4. 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. (A Stop operation will also implicitly resume a suspended process, returning it unconditionally to the ready-active state.)

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 that has caused an exception to occur and must wait for the exception condition to be processed by an exception handler. The offending process must be removed from execution in order to allow the exception-handling process to execute and to take diagnostic and, possibly, corrective action with respect to the exception condition. Therefore, the exception-wait state indicates that the offending process is waiting for action by an exception-handling process, as described further in Section 2.1.4.4. The waiting process is placed in the 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.

8. Inactive: the state of a process that has been terminated abnormally by the kernel because of an unhandled exception condition. Unlike normally terminated processes, an inactive process's PCB is not deleted but is retained on a special, "dead end" queue solely for diagnostic purposes. An inactive process cannot be reactivated. See Section 2.1.4.4.

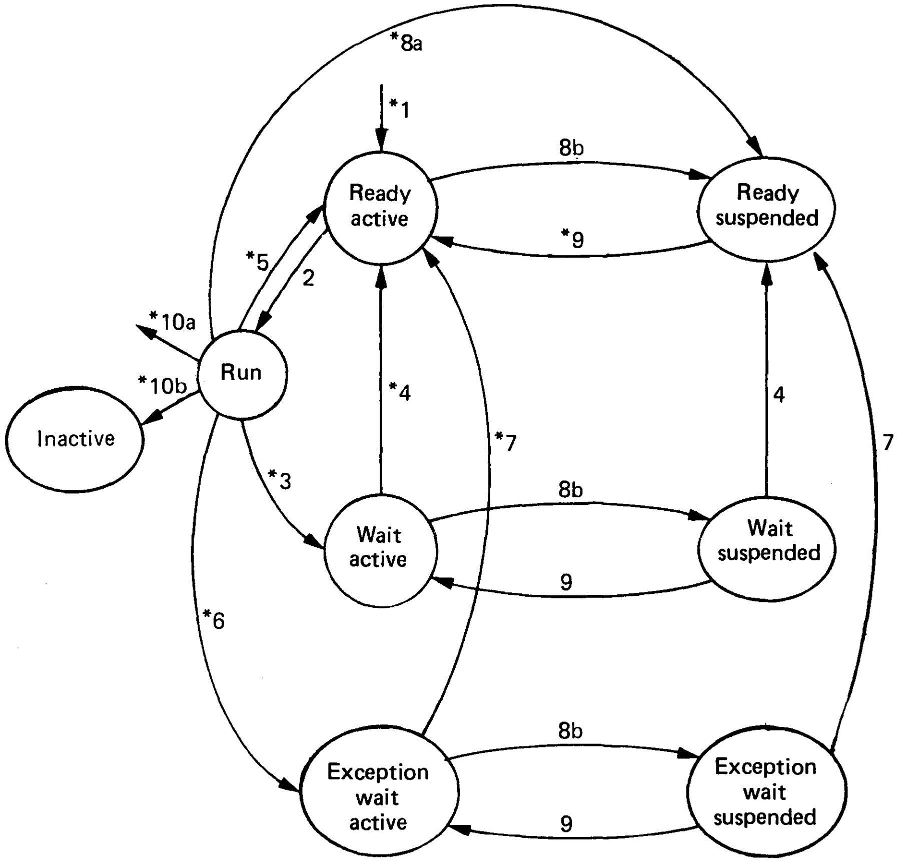
When created, a process is in the ready-active state. The possible subsequent state transitions can be summarized as follows:

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)
	Inactive (by abnormal termination)
	Nonexistent (by deletion)
Wait active	Ready active (by unblocking)
	Wait suspended (by suspension)
Ready suspended	Ready active (by resumption or forced termination)
Wait suspended	Ready suspended (by unblocking)
	Wait active (by resumption)
Exception-wait active	Ready active (by dismissal)
	Exception-wait suspended (by suspension)
Exception-wait suspended	Ready suspended (by dismissal)
	Exception-wait active (by resumption)

The inactive state is a "final" state from which no transition is possible; a process in that state is essentially nonexistent, but its context is preserved for diagnostic purposes.

Figure 2–1 shows the 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.

Figure 2–1: Process State Transitions



Legend:

*	Significant event	7	Dismissal of exception
1	Process creation	8a	Self-suspension
2	Highest-priority runnable process	8b	Suspension
3	Blocking	9	Resumption
4	Unblocking	10a	Normal process deletion
5	Preemption	10b	Process deletion for unhandled exception
6	Occurrence of exception		

MLO–391–87

2.1.3.1 Process State Codes and State Code Modifiers

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, as defined by the QUEDF$ system macro:

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 (abnormally aborted because of exception)

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.

The state-code modifier bits in the status byte of the PCB (field PC.STS) describe several possible substates in the case of a process that requires special handling during subsequent state changes. The modifier bit-mask values are represented by the following global symbols:

Modifier	Meaning
SM.FPA	Process has a pending Floating Point Accelerator exception.
SM.BCS	Blocked on complex structure. The process is blocked on multiple blocking structures because of execution of a complex primitive.
SM.ABI	Abort to inactive in progress. An unhandled exception has occurred, which causes the process to be aborted (forced to its termination entry point) as for SM.ABO. The termination will be abnormal, however, in that issuance of the Delete Process (DLPC$) request will cause the process's PCB to be placed on the inactive queue instead of being deallocated. (SM.ABO is also set whenever SM.ABI is set.)
SM.UBL	Unblocked but not yet ready. The process has been unblocked and is in the kernel resumption queue for completion of a primitive operation.
SM.ABP	Abort pending. A Stop Process (STPC$) request has been issued for this process but has not yet been honored, because the process is blocked on a ring buffer or is in an exception-wait state.

Modifier	Meaning
SM.ABO	Abort in progress. A Stop Process (STPC$) request has been issued for the process, forcing execution at its termination entry point. The process may run, block, and so on but may not be suspended. Any Suspend requests for the process will be ignored.

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 an exception-wait state is linked into one and only one logical state queue.

Conceptually, there are only five state queues, although the so-called wait queue is a logical entity that consists of many distinct queues. The characteristics of the five state 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. (The ordering of this queue 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 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. Except for the kernel's timer queue, 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. (The kernel's timer queue has a special time-dependent ordering policy.) 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.

   Normally, a blocked process is linked into a single structure's blocking queue, but the "complex" primitives (GELA$, RCVA$, WAIA$, and WAQA$) allow a process to block simultaneously on multiple structures of a given type (for example, up to four binary semaphores), and optionally on the kernel's timer queue at the same time. (Multiple blocking structures are referred to as a "complex structure"; the queuing for that case is handled by the complex-structure-descriptor field in the process's PCB.)

5. Inactive queue: a doubly-linked list of all inactive PCBs, ordered according to process priority. (The ordering of this queue is immaterial, since PCBs are never removed from it.) The kernel global symbol $IACTV identifies the list head, in kernel data space, for this queue.

No state queue exists 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.

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. That can happen if either the running process or an ISR performs an operation that unblocks a wait-active process, resumes a ready-suspended process, or "dismisses" an exception-wait-active process. Preemption can also occur if the running process creates a new, higher-priority process, lowers its own priority, or raises the priority of another process.

Many kinds of semaphore and ring buffer operations (for example, a signal operation) 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.

When a process is preempted, it is always placed at the head of the ready-active queue, before any other processes of equal priority already on the queue. Thus, a preempted process always has the highest effective scheduling priority relative to any other ready-active process of the same priority. That is contrary to the equal-priority queuing policy effective for other operations, such as unblocking, which places a newly queued process behind any other processes of the same priority already on the queue.

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 counting semaphore:

  WAIT procedure (WAIT$ request)
  WAIT_ANY procedure (WAIA$ 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 (WAIQ$ request)
  GET_PACKET_ANY procedure (WAQA$ request)
  RECEIVE procedure (RCVD$ request)
  RECEIVE_ANY procedure (RCVA$ request)
  RECEIVE_ACK procedure (RCVD$ request)
  RECEIVE_ANY_ACK procedure (RCVA$ request)

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

- An unconditional Get Element or Put Element operation on a ring buffer:
  GET_ELEMENT procedure (GELM$ request)
  GET_ELEMENT_ANY procedure (GELA$ 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.

- A Sleep operation:
  SLEEP procedure (SLEP$ request)

  Blocking condition: The time interval specified in the Sleep request has not yet expired.

The conditional forms of the operations listed previously (for example, the COND_WAIT function or the WAIC$ request) never cause the executing process to block.

A blocked process is unblocked either by a primitive operation that provides the signal or resource for which the process was waiting or by elapse of a given time interval for a process blocked on the kernel timer queue. Unblocking implies a transition from 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 of the running process. The following primitive operations may unblock a waiting process:

- A Signal operation on a binary or counting semaphore:
  SIGNAL procedure (SGNL$ request)
  COND_SIGNAL function (SGLC$ request)
  SIGNAL_ALL procedure (SALL$ request)

- A Put Packet or Send Data operation on a queue semaphore:
  PUT_PACKET procedure (SGLQ$ request)
  COND_PUT_PACKET function (SGQC$ request)
  SEND procedure (SEND$ request)
  COND_SEND function (SNDC$ request)
  SEND_ACK procedure (SEND$ request)
  COND_SEND_ACK function (SNDC$ request)

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

  GET_ELEMENT procedure (GELM$ request)
  GET_ELEMENT_ANY procedure (GELA$ request)
  COND_GET_ELEMENT function (GELC$ request)
  PUT_ELEMENT procedure (PELM$ request)
  COND_PUT_ELEMENT function (PELC$ request)

  Unblocking conditions: A Get Element operation will unblock a process waiting to put elements into the same buffer if the Get 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 the Put supplies enough elements to satisfy the requirements of the Get operation.

When the operation is successful, the conditional form of the semaphore operations listed previously 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 active process by requesting a Suspend (SPND$) 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 between the active and suspended substates.

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 a 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 SPND$ and RSUM$ primitives in Chapter 3.)

2.1.4.4 Exception Handling

The MicroPower/Pascal kernel reports certain processor traps as exception conditions that can be intercepted by an exception-handling process. The following processor traps are so reported:

Vector	Name and Description
000	Vector fetch trap: SBC–11/21 and LSI–11/23–PLUS only
004	Trap to 4: Bus timeout; 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
140	Break trap to 140: SBC–11/21 only
244	Floating-point exception: FP–11, FIS, or FPA option
250	Memory-management unit error: MMU option in effect

The kernel also reports a stack overflow or underflow exception for user-stack boundary violations detected by the kernel during process context switching.

In addition, a large set of software exceptions are defined for other error conditions detected by software, whether at kernel, system process, or user process level. Except for the kernel-detected stack boundary violations, however, these conditions are not automatically reported as exceptions by the kernel. Rather, such conditions must be reported as such at process level, by means of the Report Exception (REXC$) primitive, by the process that itself detects the error or receives an error indication from the kernel or a system process. (The MicroPower/Pascal OTS provides optional, automatic exception reporting for processes implemented in Pascal.) Table 7–1 lists all exception types and codes.

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 is 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 through its exception queue semaphore, for disposition according to the management strategy implemented by that handler. The handler can dismiss the exception, pass the exception to the process's exception service routine or procedure, if any, or request that the process be aborted. See the DEXC$ request in Chapter 3 for more 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:

- The process is reentered at its exception service routine (or Pascal exception service procedure), if any. The process stack will contain a frame of information related to the exception condition.
- If no exception service routine or procedure has been established, the kernel sets special state-code modifier bits in the PCB field PC.STS, indicating an abnormal-abort substate (SM.ABI and SM.ABO), and then forces the process to its termination entry point, as if a Stop Process (STPC$) request had been issued for the process. However, because of the special substate when the process issues its Delete Process (DLPC$) request (which customarily ends a termination routine), the process's PCB is not deleted but is placed on the inactive queue. The process is essentially terminated, but its final context is preserved, including an exception code stored in the PCB (field PC.ESC).

If an exception handler does exist for the faulting process and its disposition of the process is "abort," the kernel also sets the SM.ABI state code modifier as for the case just described.

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

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. The three categories of significant events are as follows:

- A primitive executed by the running process that causes it to leave the run state, typically switching to the wait-active state (blocking)
- A primitive executed by either the running process or an interrupt service routine that causes another process to enter the ready-active state, typically by unblocking, which in turn may cause preemption of the running process
- Occurrence of an exception condition that is dispatched to an exception-handling process, causing the running process to enter the 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, the scheduler 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. The scheduler also performs a process context switch, saving and restoring the context of the old and new running processes, so the latter gains control of the CPU on return from kernel processing.

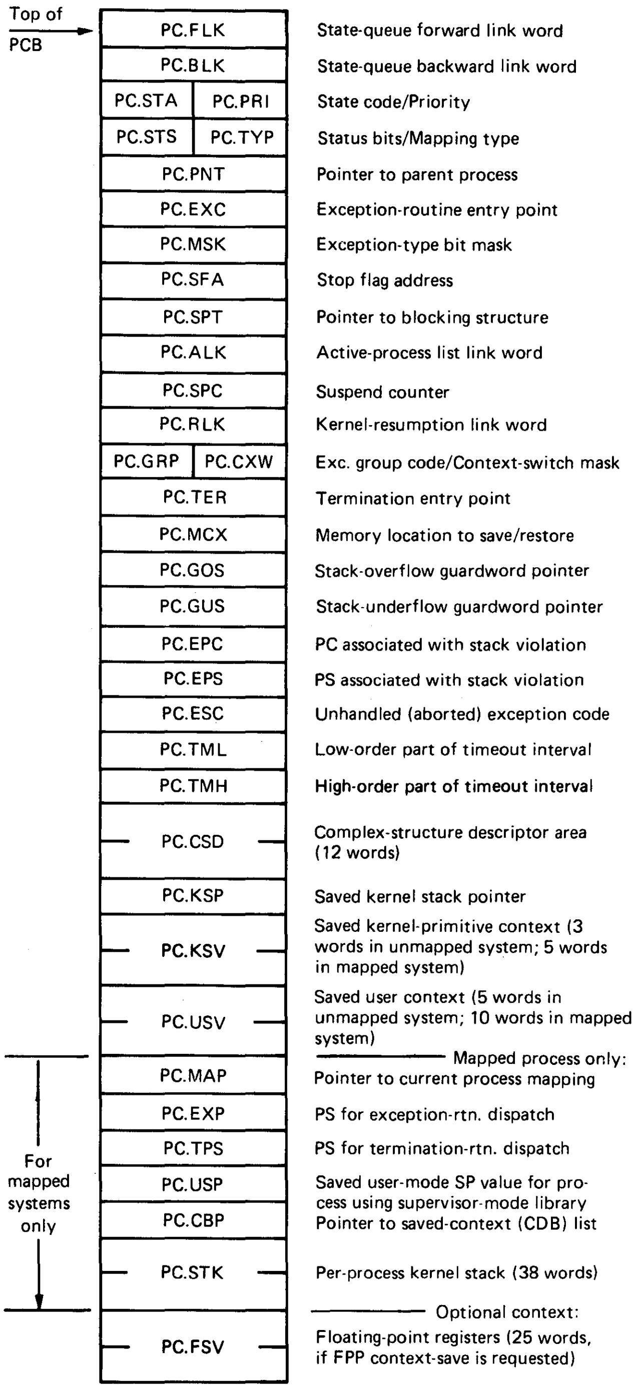
2.1.5 Process Control Block (PCB)

A process is physically represented within the kernel by a process control block (PCB), the system data structure that identifies a particular activation of an 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:

- Defines the name, if any, and all other fixed attributes of the process.
- Contains all dynamic state information maintained by the kernel about the process. This collection of information is called the software context of the process.
- Provides a save area for kernel context that must be saved on a per-process basis under certain circumstances.
- 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 the run state. (In an unmapped system, the R3, R4, R5, PC, and PS values are saved in an interrupt stack frame on the process stack rather than in the PCB whenever the process is either interrupted or switched out of run state.) The PCB for a mapped process also points to a separate save/restore area for the process's MMU register contents, which is initialized during process creation. A process's mapping context is optionally saved in this area when the process is switched out and is always restored from this area whenever the process is switched in.

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. Figure 2–2 shows how the PCB is organized.

Figure 2–2: Process Control Block (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-routine entry point
PC.MSK	Exception-type bit mask
PC.SFA	Stop flag address
PC.SPT	Pointer to blocking structure
PC.ALK	Active-process list link word
PC.SPC	Suspend counter
PC.RLK	Kernel-resumption link word
PC.GRP PC.CXW	Exc. group code/Context-switch mask
PC.TER	Termination entry point
PC.MCX	Memory location to save/restore
PC.GOS	Stack-overflow guardword pointer
PC.GUS	Stack-underflow guardword pointer
PC.EPC	PC associated with stack violation
PC.EPS	PS associated with stack violation
PC.ESC	Unhandled (aborted) exception code
PC.TML	Low-order part of timeout interval
PC.TMH	High-order part of timeout interval
PC.CSD	Complex-structure descriptor area (12 words)
PC.KSP	Saved kernel stack pointer
PC.KSV	Saved kernel-primitive context (3 words in unmapped system; 5 words in mapped system)
PC.USV	Saved user context (5 words in unmapped system; 10 words in mapped system)
	Mapped process only:
PC.MAP	Pointer to current process mapping
PC.EXP	PS for exception-rtn. dispatch
PC.TPS	PS for termination-rtn. dispatch
PC.USP	Saved user-mode SP value for process using supervisor-mode library
PC.CBP	Pointer to saved-context (CDB) list
PC.STK	Per-process kernel stack (38 words)
	Optional context:
PC.FSV	Floating-point registers (25 words, if FPP context-save is requested)

MLO–392–87

Table 2–1 describes the PCB fields shown in Figure 2–2. The fields noted as dynamic reflect the current state of the process and constitute its dynamic context.

Table 2–1: PCB Field Descriptions

Field	Description
PC.FLK	Forward pointer to the next PCB in the current state queue; dynamic (used for linking into the kernel timer queue if the PCB is blocked on a complex structure through field PC.CSD)
PC.BLK	Backward pointer to the previous PCB in the state queue; dynamic (zeroed when PCB is linked into a blocking queue)
PC.PRI	Process priority value (range 0 to 255); set during process creation; may be modified by the CHGP$ 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 SERA$ primitive (Chapter 3)
PC.MSK	Bit mask of exceptions that the process will accept; set by SERA$
PC.SFA	Address of process's stop flag, if any; set by the SSFA$ primitive (Chapter 3)
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 unterminated processes; dynamic
PC.SPC	Suspend count; modified by the SPND$ and RSUM$ primitives (Chapter 3); dynamic
PC.RLK	Pointer to the next PCB in the kernel resumption list; dynamic (used by the kernel to queue processes awaiting "kernel resumption" following certain unblocking operations)
PC.CXW	Context-switch option bits; set during process creation (see CRPC$ or DFSPC$ primitive in Chapter 3)
PC.GRP	Exception group code; set during process creation (see CRPC$ or DFSPC$)
PC.TER	Termination entry point; set during process creation
PC.MCX	Address of optional user-memory location to be saved in PC.USV; zero value if CX$MCX option was not selected; set during process creation

Table 2-1 (Cont.): PCB Field Descriptions

Field	Description
PC.GOS	Lower-boundary guardword address for stack overflow checking; set during process creation
PC.GUS	Upper-boundary guardword address for stack underflow checking; set during process creation
PC.EPC	PC save/restore word for kernel-detected stack guardword violation, used by stack-exception reporting mechanism; dynamic
PC.EPS	PS save/restore word for kernel-detected stack guardword violation, used by stack-exception reporting mechanism; dynamic
PC.ESC	Exception code corresponding to an unhandled exception that caused the process to be aborted; valid only when process state is SC.IAC (inactive); dynamic
PC.TML	Low-order portion of the time-out interval value maintained by the kernel's clock service routine when a process is blocked on the kernel timer queue; set by the SLEP$ primitive or a complex primitive and modified by the kernel clock ISR; dynamic
PC.TMH	High-order portion of the time-out interval doubleword formed by PC.TML and PC.TMH; dynamic
PC.CSD	The complex structure descriptor area used by the WAIA$, WAQA$, RCVA$, or GELA$ primitive when a process is waiting on multiple blocking structures (field PC.FLK is the link into the kernel timer queue if the process is also blocked for timeout); dynamic
PC.KSP	Saved stack pointer for resumed kernel-primitive operations; this value points into the process stack in an unmapped system or into PC.STK in a mapped system (see note below); dynamic
PC.KSV	Save area for kernel-primitive context: in an unmapped system, three words for R4, R3, and R0; in a mapped system, five words for R4, R3, R0, and kernel-mode PARs 2 and 3 (see note below); dynamic
PC.USV	Save area for user-context switch: in an unmapped system, 5 words for user SP, R0, R1, R2, and the optional memory location; in a mapped system, 10 words for previous-mode SP, user-mode R0-R5, PC, PS, and the optional memory location; dynamic
PC.MAP[1]	Pointer to the 16- or 32-word save /restore area for the process's current memory-mapping register (PAR and PDR) values; set during process creation. The MMU registers are dynamically saved in the area on switchout only if the CX$KT option was specified, indicating that the privileged or driver-mapped process modifies its mapping itself rather than through a primitive service. The MMU registers are always restored from this area on switchins, however. (The size of the save area is reflected by the setting of the CX$IAD bit, which is determined at build time.)
PC.EXP[1]	PS used for dispatching to the exception routine, significant if a supervisor-mode shared library is in effect; dynamic.

[1] Present only in mapped systems

Table 2–1 (Cont.): PCB Field Descriptions

Field	Description
PC.TPS[1]	PS used for dispatching to the termination routine, significant if a supervisor-mode shared library is in effect; dynamic.
PC.USP[1]	User-mode stack pointer saved and restored in case the process accesses a supervisor-mode shared library (an LSI–11/73 target build option); dynamic.
PC.CBP[1]	Pointer to a pushdown list of context descriptor blocks (CDBs), containing explicitly requested "snapshots" of process mapping, created by the SCTX$ primitive and used by the RCTX$ primitive; zero if the list is empty; dynamic.
PC.STK[1]	Per-process kernel stack; 38 words; in a mapped system, the kernel uses this area as its stack for primitive operations; dynamic (in an unmapped system, the process stack is used instead).
PC.FSV[2]	Save area for FP–11 floating-point registers; 25 words for processes that use FP–11 floating-point instructions (KEF11–A or FPF11 option).

[1] Present only in mapped systems

[2] Present only if the CX$FPP option was selected for the process, indicating that it uses the FP–11 floating-point processor

Note

> The PC.KSP and PC.KSV context values are valid while a process is blocked on a structure other than a single binary or counting semaphore. The kernel uses these context values when a primitive service must resume operation in order to unblock the waiting process and switch it to its subsequent ready state. In contrast, the user-context (PC.USV) values are valid whenever the process is not in the run state.

The size of a PCB varies both by hardware environment and by floating-point processor (FP–11) usage, as follows:

- For a process in an unmapped system:
  Without FPP context, 43 words
  With FPP context, 68 words
- For a process in a mapped system:
  Without FPP context, 93 words
  With FPP context, 118 words

  In addition, the MMU restore area pointed to by PC.MAP adds 16 or 32 words to the space requirement for each PCB (32 words only if separate data-space mapping is in effect for the process: bit CX$IAD = 1 in field PC.CXW).

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 any 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.

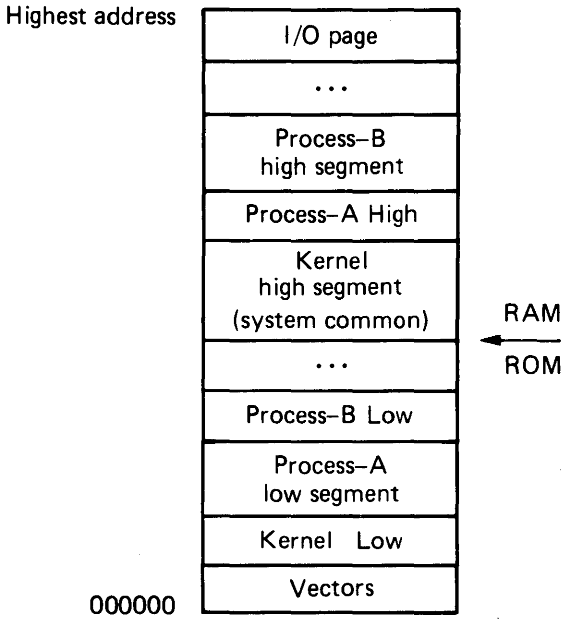
2.1.6 Memory Partitioning and Process/Program Segmentation

MicroPower/Pascal uses a memory-layout technique designed to work effectively in either a ROM/RAM hardware environment or a RAM-only environment. If a target system does include both ROM and RAM, enough ROM must be configured in low memory to contain the vector area and at least the kernel's pure-code and pure-data segment. (The low-ROM requirement does not apply in the special case of a CMR21 target system.) From that point on, the ROM and RAM areas may in principle be configured as desired. The physical addresses implemented by the memory configuration need not be contiguous; "holes" may exist in memory both within and between the ROM and RAM areas. Also, ROM and RAM may be interspersed in physical memory; that is, some ROM may be configured at higher addresses than RAM.

As a practical consideration, either fragmented memory or interspersed ROM and RAM can cause some memory to be wasted, may necessitate a complicated, nonautomatic application building procedure, or both. In a ROM/RAM target system, you should configure all ROM as low memory and all RAM as high memory if at all possible, since at least some ROM must be low addressed, as already stated. With a FALCON–PLUS target, however, the memory-map 1 configuration may be required for a given application and necessitate building for interspersed ROM/RAM.

To allow the MicroPower/Pascal build utilities to handle ROM/RAM as well as mapped or unmapped RAM-only applications, the address space of a MicroPower/Pascal process family must be partitioned into two segments, low and high, through appropriate program sectioning. (For processes implemented in Pascal, program sectioning is automatic.) 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 at least two physically separate memory regions in a ROM/RAM environment, as shown in Figure 2–3 for the simple low-ROM and high-RAM case. The kernel's address space is partitioned in the same manner. (For simplicity, each process family is represented as a single process.)

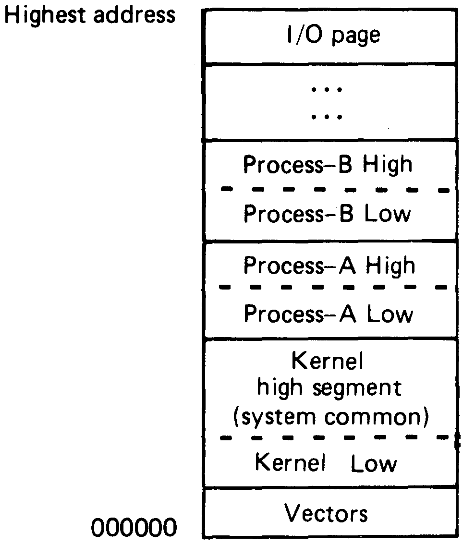
Figure 2–3: ROM/RAM Physical Memory Layout



MLO–393–87

In a RAM-only system, the pure (low) and impure (high) segments are not physically separated in memory (ignoring the small gaps imposed by the memory-mapping hardware in a mapped environment). The RAM-only memory layout is shown schematically in Figure 2–4.

Figure 2-4: RAM-Only Physical Memory Layout



MLO-394-87

To achieve the low/high process code and data segmentation described above, a MACRO-11 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.

As previously stated, program sectioning is provided transparently by the MicroPower/Pascal compiler for a Pascal program. The MACRO-11 programmer can use the PURE$, PDAT$, and IMPUR$ program sectioning macros; preceding code; and pure-data and impure-data sequences, respectively, to conveniently generate the proper program sectioning directives for a process that can be handled automatically by the build utilities. (See Chapter 3.)

If a process has special relocation and memory-allocation requirements, however, additional program sectioning directives may be needed. (The RELOC and MIB utilities have options that permit special relocation and memory allocation by p-sect name.) Nondefault relocation might be required, for example, for most efficient building of an application for an unmapped, interspersed ROM/RAM target.

During the application build cycle, the RELOC utility groups a process's p-sects according to the read-only versus read/write attribute. Within each group, the p-sects are sorted into alphabetical order by p-sect name. One important effect of the RO versus RW p-sect grouping and the subsequent alphabetical sorting is to make sure that the critical read-only p-sect .ALST. always appears first in a static-process memory image, as required by the MIB utility and the kernel. (See the DFSPC$ macro description in Chapter 3.)

In an unmapped environment, all processes have direct access to kernel memory space and in particular to the kernel's impure segment. The total application is limited to 28K words in a target system with a 4K-word I/O page or to 30K words in a target system with only a 2K-word I/O page and no high-memory firmware, such as an SBC-11/21 target. (If an MVS11-DD or MVS11-ED memory module is used with the "extra 2K words" option enabled, 30K words of usable memory is possible. The option effectively halves the size of the I/O page for an LSI-family target.)

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 124K words of usable memory, and a single process or process family with general mapping can occupy up to 32K words. A mapped system with 22-bit addressing can support up to approximately 2M words of memory. If the target system provides separate I&D-space mapping, as does the LSI–11/73, a general-mapped process family can occupy up to 64K words (32KW code and 32KW data) if the mapping separation is used.

2.1.7 Process Mapping Types

The information in this section applies only to a mapped-memory hardware environment, such as an LSI–11/23 target system. For simplicity, assume that I&D-space separation is not in effect except where specifically noted.

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), run-time shared libraries, and four types of processes: general, device access, driver, and privileged. More specifically, a mapping type identifies a particular active page register (APR) usage convention associated with one of these objects. Both kernel mapping and ISR mapping use the kernel-mode set of APRs.

A run-time shared library in a mapped system can be mapped in either user mode or supervisor mode. A user-mode shared library uses the user-mode set of APRs and thus does not have a mapping of its own, strictly speaking, but affects the mapping of any process that is built with such a library. In a target system that supports supervisor mode, a supervisor-mode shared library uses the supervisor-mode set of APRs for mapping library code and pure data. Thus, a supervisor-mode shared library does have its own mapping, independent of that of a referencing process, but use of such a library also affects the mapping of the referencing process to a limited extent, imposing a special restriction on user-mode APR 0.

The four types of process mappings use the user-mode set of APRs. (Figures 2–5 to 2–11, discussed later, show the APR assignments for each kind of mappable object and for each of the process mapping types.)

A mapping type is specified when a static process is defined. Any subprocesses created by the static process inherit its mapping type and its mapping register values, since all the code and data associated with a given process family resides in the same virtual address space. (Although a process's mapping type is fixed, a process can dynamically modify its mapping register values without affecting the mapping of any other process in the family, since the context of every process includes a unique mapping image that is restored on each context switch.) The basic characteristics of the general, device-access, driver, and privileged process mappings are as follows:

- 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 access 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) is available for process code and data. Therefore, the pure and impure segments defined for a static process and its subprocesses, if any, can occupy up to 32K words. (If I&D-space separation is available on the target system, the process family size can range up to 64K words, with the full range of virtual addresses available both for the pure-code segment and for the pure- and impure-data segments.)

Because of 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 requirement is enforced by the build utilities. Thus, a sizable "hole" in the virtual address space (up to 4K–32 words) may exist between the highest address in the low segment and the beginning of the high segment, reducing the potential process family size by that amount. (This is true for processes of any mapping type.) If I&D-space separation is in effect, the possible hole will exist only in the process's data address space, between the pure-data and impure-data segments required by such separation.

In Pascal, if no mapping attribute (DEV_ACCESS, DRIVER, or PRIVILEGED) is specified, the process family defined by the program has general mapping.

- Device access: intended for processes that require access to the I/O page (for example, to device CSRs), 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. Device-access mapping differs from general process mapping only in that virtual addresses 160000 to 177777 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 28K words. (If I&D-space separation is available on the target system, the process family size can range up to 60K words, with the full 32KW range of virtual addresses available for the pure-code segment and 28K words of virtual address space for the pure- and impure-data segment.)

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

- Driver: intended for device-handling processes that include an ISR. Driver mapping allows direct access to system data structures (to the kernel's common data space) as well as to the I/O page. Driver mapping also allows APR 1 to be used as a "stratch" address register (for example, for mapping to another process's input or output data buffer area). Driver mapping restricts process size to a maximum of 8K words but allows very efficient queue semaphore operations, for interprocess message transmission, and is fully compatible with the kernel-mode mapping of an ISR. Although I&D-space separation, where available, is possible for driver process-level code and data (but not for the ISR code and data), a properly designed driver process family is not likely to require such separation.

  The lowest 4K words of virtual address space should not be used. The next 4K words of virtual space (addresses 020000 to 037777) are initially unmapped and are available for any dynamic use (typically for mapping to a requesting process's buffer space). Virtual addresses 040000 to 077777 (8K words) are available for statically allocated driver process/ISR code and data. Virtual addresses 100000 to 157777 are mapped as needed to the kernel's common data area (variable in size up to 12K words), and addresses 160000 to 177777 are mapped to the I/O page (4K words).

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

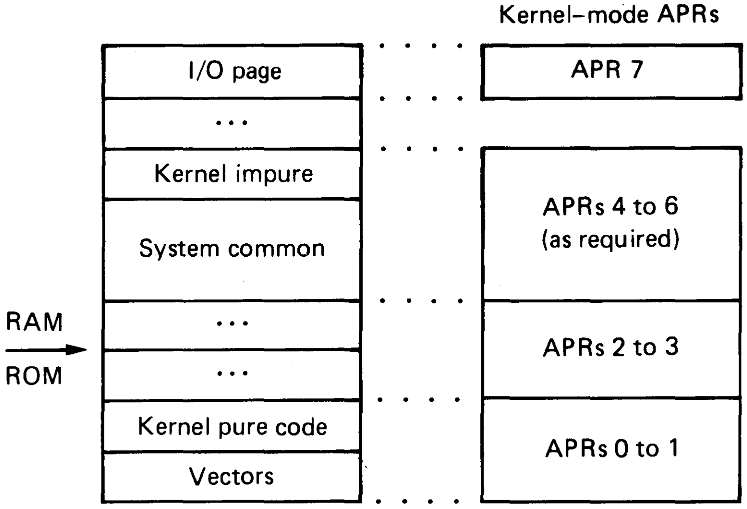
- Privileged: intended for processes that need direct access to system data structures (to the kernel's common data space) as well as to the I/O page. Also called full-system mapping, privileged mapping restricts process size to a maximum of 16K words but allows very efficient queue semaphore operations, for interprocess message transmission. Privileged mapping is commonly used by processes that provide systemwide services other than device handling and is typically used by exception handlers, which generally require direct access to PCBs.

  The lowest 16K words of virtual address space (addresses 0 to 077777) are available for process code and data. Virtual addresses 100000 to 157777, corresponding to APRs 4, 5, and 6, are mapped as needed to the kernel's common data area (variable in size up to 12K words). (Any of those APRs can be borrowed for dynamic remapping. APR 6 in particular will effectively be a scratch APR in an application in which the kernel's data area does not exceed 8K words.) Virtual addresses 160000 to 177777 are mapped to the I/O page (4K words). (If I&D-space separation is available on the target system, the process family size can range up to 48K words, with the full 32KW range of virtual addresses available for code, and 16K words of address space available for pure and impure process data.)

  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.

Figure 2–5: Kernel Mapping



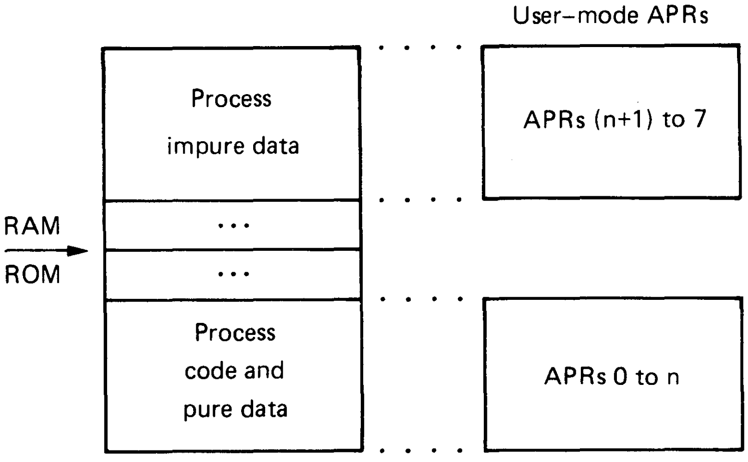
Kernel context: PC, PS, SP, R3 to R5, Kernel APRs 2 and 3.

MLO–395–87

Kernel-mode APRs 0 and 1 map the hardware vector area and the kernel pure-code segment; thus, the latter is limited to less than 8K words. APRs 2 and 3 are scratch address registers; they are modified as needed to map to user address space (for example, mapping user argument blocks). APRs 4 to 6 are used as needed to map system-common memory and the kernel's own impure data, allowing up to 12K words of system data. Only APR 4 or APRs 4 and 5 may actually be in use, depending on the amount of memory allocated for the system-common area in the RESOURCES configuration macro (see Chapter 4) and on the number of interrupt vectors in use. (The system-common area consists of two separately configurable memory pools in which the kernel allocates space for dynamic data structures—such as semaphores, ring buffers, and PCBs—and for queue packets.) Privileged and driver processes can access the entire kernel data segment, which is mapped by user-mode APRs 4 to 6 for those kinds of processes. See Section 2.2.4 for further information on kernel data segment organization. Kernel-mode APR 7 maps the I/O page. (I&D-space separation is not applicable to the kernel.)

Figure 2–6 illustrates general process mapping, which provides access only to user-defined memory. Note that Figure 2–6 does not reflect possible I&D-space separation.

Figure 2–6: General Process Mapping



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

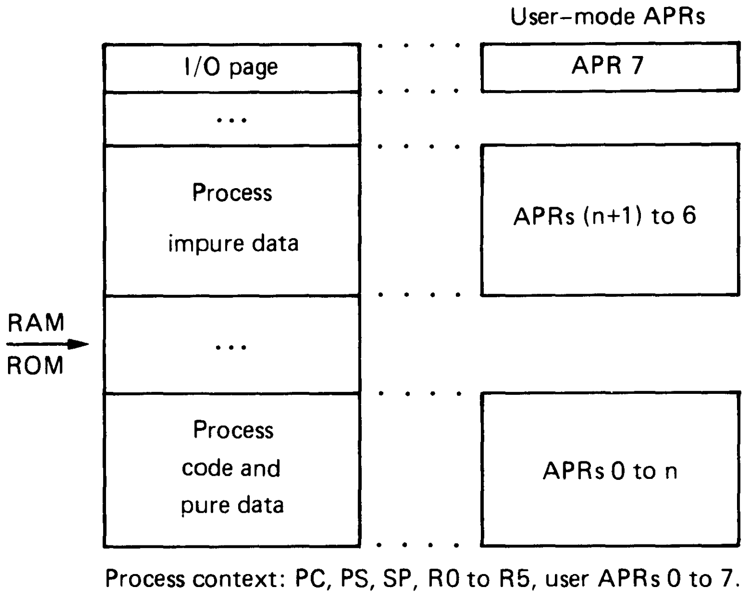
MLO-396-87

All eight user-mode APRs are available for mapping process code and data, allowing a maximum of 32K words of process space. Process pure code is mapped by APRs 0 to n, where n <7, allowing 4K*(n+1) words of code (up to 28KW). Process data is then mapped by the remaining APRs, (n+1) through 7, permitting 4K*(7–n) words of data.

I&D-space separation, available on some target systems, potentially doubles the virtual address space available to a process—at the possible expense of additional context switching time. If that separation is in effect for a given process, two full sets of user-mode APRs exist—one for code and one for data. Thus, the process's pure-code segment can consist of up to 32K words, and its combined pure- and impure-data segments can consist of up to 32K words, less a possible hole of up to 4K words in the data-space virtual addresses in the ROM/RAM case.

Figure 2–7 illustrates device-access process mapping, which provides access to the I/O page but not to the system-common area. Note that Figure 2–7 does not reflect possible I&D-space separation.

Figure 2–7: Device-Access Process Mapping



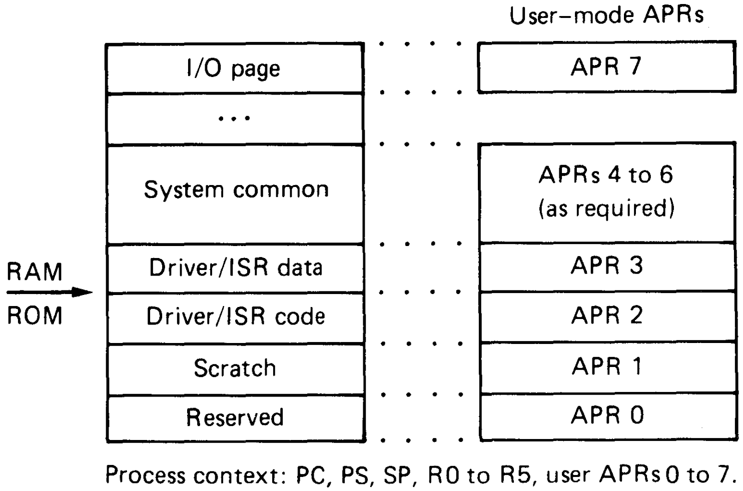
MLO–397–87

User-mode APRs 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 APRs 0 to n, where n <6, allowing 4K*(n+1) words of code (up to 24KW). Process data is then mapped by the remaining APRs, (n+1) to 6, permitting 4K*(6–n) words of data. APR 7 maps the I/O page.

If supported by the target system, I&D-space separation may be used for a device-access process as well as for a general mapped process. This permits a maximum code segment size of 32K words, mapped by I-space APRs 0 to 7, and a maximum of the pure- and impure-data segment sizes, combined, of 28K words, mapped by D-space APRs 0 to 6. (The possible virtual-address break between the pure- and impure-data segments in an application built for ROM/RAM can cause a hole of up to 4K words in the data-space virtual addressing.) Data space APR 7 maps the I/O page.

Figure 2–8 illustrates driver-process mapping, which provides access to both the I/O page and the system-common area.

Figure 2–8: Driver Memory Mapping



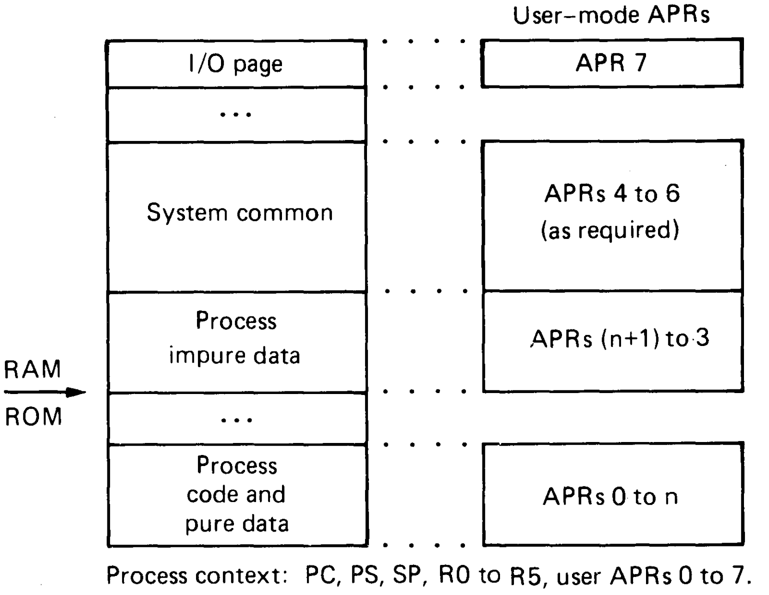
MLO-398-87

User-mode APR 1 is a scratch register for the driver process-level code. (The ISR code is mapped by kernel-mode APRs at interrupt level, and kernel-mode APR 1 is not scratch but can be borrowed. See Figure 2–10.) APRs 2 and 3 map the combined process/ISR code and data, respectively, which can occupy up to 8K words. APRs 4 to 6 map the system-common data area, as needed. APR 7 maps the I/O page. User-mode APR 0 is reserved by DIGITAL for future device driver interfaces.

I&D-space separation is possible but not generally applicable to a driver mapped process and in particular is not valid for the ISR code and data that is normally included in the process code and data segments.

Figure 2–9 illustrates privileged-process mapping, which provides access to both the I/O page and the system-common area. Note that Figure 2–9 does not reflect possible I&D-space separation.

Figure 2–9: Privileged Process Mapping



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

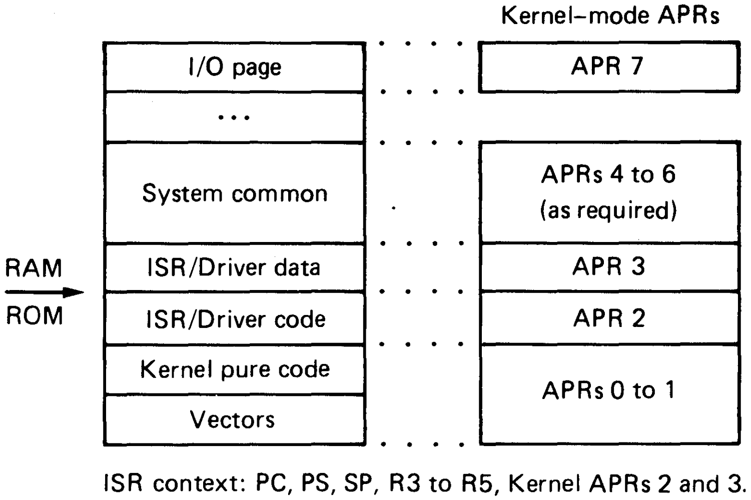
MLO–399–87

User-mode APRs 0 to 3 map the process code and data, which may total 16K words. APRs 4 to 6 map the system-common data area, as needed. APR 7 maps the I/O page.

If supported by the target system, I&D-space separation may be used for a privileged process, significantly increasing the potential maximum process size at a possible cost in performance. The separation permits a maximum code segment size of a full 32K words, mapped by I-space APRs 0 to 7, and a maximum of the pure- and impure-data segment sizes, combined, of 16K words, mapped by D-space APRs 0 to 3. (The possible virtual-address break between the pure- and impure-data segments in an application built for ROM/RAM can cause a hole of up to 4K words in the D-space virtual addressing.) D-space APRs 4 to 6 map the system-common data area, and D-space APR 7 maps the I/O page.

Figure 2–10 illustrates ISR mapping; ISRs are described in Chapter 7.

Figure 2–10: Interrupt Service Routine Mapping

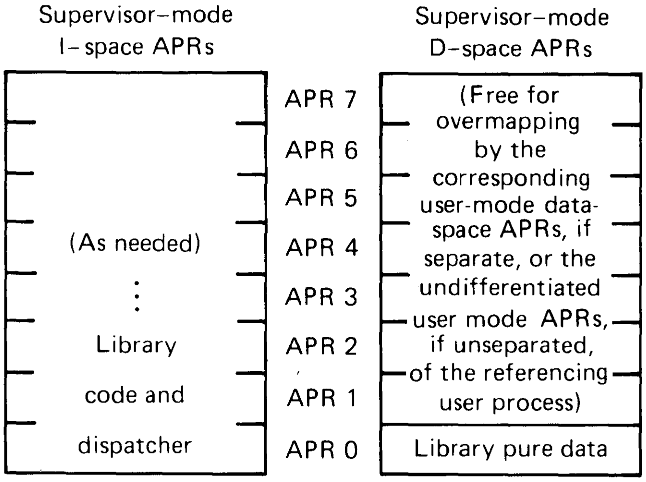


MLO–400–87

The mapping of ISRs uses the kernel-mode APRs and is designed to be very fast. Kernel-mode APRs 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. APR 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, APR 1 must, if borrowed, be restored before issuing a FORK$ request and any subsequent primitive requests. Note that I&D-space separation is not applicable to an ISR.

Figure 2–11 illustrates supervisor-mode shared-library mapping.

Figure 2–11: Supervisor-Mode Shared-Library Mapping


Supervisor-mode I-space APRs
Supervisor-mode D-space APRs
(As needed)
⋮
Library code and dispatcher
APR 7
APR 6
APR 5
APR 4
APR 3
APR 2
APR 1
APR 0
(Free for overmapping by the corresponding user-mode data-space APRs, if separate, or the undifferentiated user mode APRs, if unseparated, of the referencing user process)
Library pure data

(A supervisor-mode shared library always uses I&D-space separation, regardless of whether that separation is in effect for a referencing process.)

MLO–401–87

The run-time library pure-code segment, comprising the library dispatcher and subroutine code, is mapped by supervisor-mode I-space APRs 0 through n as required. Thus, the library code in no way impinges on the virtual address space of a referencing process.

The library's pure-data segment is mapped by supervisor-mode D-space APR 0. (A shared library contains no impure-data segment of its own.) The remainder of the library's D-space APRs are overmapped with process-mapping values each time a referencing process is switched into the run state. That is, except for APR 0, a calling process's data-space mapping, if separated (or undifferentiated I&D-space mapping, if not separated) is copied to the library's D-space APRs 1 through 7 to allow access to the process's data by the library routines. That implies, of course, that the caller's D-space APR 0 (or its undifferentiated APR 0) maps no process data that the library routines need to access, since the library's D-space APR 0 is reserved for its own read-only data and is never modified. Therefore, the build-time implications for a static process that references a supervisor-mode shared library are the following:

- For a static process built with I&D-space separation, the RELOC utility will by default start the process's pure-data segment at virtual address 20000, thus removing the D-space APR 0 from the process's mapping.
- For a static process built without I&D-space separation, the RELOC utility will by default start the process's pure-code (low) segment at virtual address 20000, thus removing the undifferentiated APR 0 from the process's mapping.

- For a static process built without I&D-space separation, you may force your mapping to begin at virtual address 0 by using the RELOC utility option /RO:0 (VMS/RSX) or /O:0 (RT–11) at build time. That lets you retain the use of APR 0 (assuming that APR 0 maps only code or that none of the pure data, if any, also mapped by APR 0 is accessed by the library routines). In any case, if use of APR 0 is forced, it must not map any read/write data. Violation of either constraint is likely to result in unpredictable and probably very obscure run-time errors. (In an all-RAM target environment, the process's code and pure-data segments are brought together and by default are contiguous with the impure-data segment in virtual and physical space.) The process's read/write data can be separated from the code and pure data, as if for a ROM/RAM target, through use of the /AL (VMS/RSX) or /X (RT–11) RELOC option, which forces the impure-data segment to the next available 4KW virtual address boundary, satisfying the read/write data constraint. For a static process comprising at least 4KW of code, conformance to the more general constraint on APR 0 is implicit. For a Pascal static process having a code segment smaller than 4KW, total code/data separation can be achieved by using the RELOC option /QB:.IDAT.:20000 (under RSX/VMS) or /Q (under RT–11) supplying the .IDAT. and 20000 values in response to a RELOC interactive prompt. (If you are using the MPBUILD or MPBLD facility for application building, you can edit the generated build-command file to add the required RELOC options.)

In general, for simplicity of application building and avoidance of programming constraints, I&D-space separation should be used when building a user process with a supervisor-mode shared library. The automatic relocation in that case, reserving the low-order 4KW of virtual data-space addressing, still allows up to 28K words of program data for a general mapped process, up to 24KW for device-access mapping, or up to 12KW for privileged mapping.

2.2 System Data Structures

The MicroPower/Pascal run-time system uses a variety of dynamic data structures, which 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 those structures. However, you do not need to know how they are implemented in order to use the kernel services; the primitive-request interface hides this level of detail. The 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 typed data structures (for example, semaphores and ring buffers), message packets, and 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 offset symbols and other MACRO–11 symbols shown in this section are defined by the QUEDF$ macro in the COMM and COMU macro libraries except where indicated otherwise. The overall organization of the system-common memory area is also described.

Several kernel structures related to exception dispatching and interrupt dispatching are described in Chapters 6 and 7.

2.2.1 Typed Data Structures

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

- Binary semaphore (BSM)
- Counting semaphore (CSM)
- Queue semaphore (QSM)
- Ring buffer (RBF)
- Shared region descriptor (SRD)
- Logical-name value (LNM)
- Process control block (PCB)
- Unformatted structure (UDF)

Semaphores, ring buffers, and unformatted structures are explicitly created and deleted by the Create Structure (CRST$) and Delete Structure (DLST$) primitives. Logical-name structures are created and deleted by the Create Logical Name (CRLN$) and Delete Logical Name (DLLN$) primitives. Shared region descriptors are created and deleted by the Create Shared Region (CRSR$) and Delete Shared Region (DLSR$) primitives. PCBs are implicitly created and deleted as a part of process creation and deletion. Note that the PCB was defined in Section 2.1.5 under the general discussion of processes.

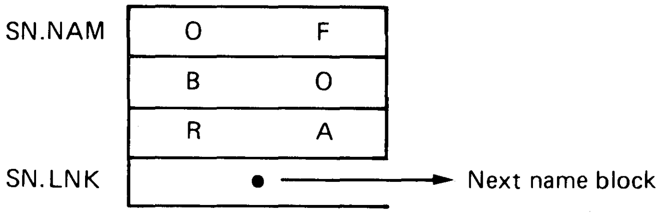
No kernel operations other than creation and deletion are defined on an unformatted structure; its internal format is undefined. This type of structure is available for application-defined purposes.

All typed structures can be named. Section 2.1.2 discusses the naming of PCBs.

2.2.1.1 Structure Names and Name Blocks

The kernel allows a run-time 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 typed structures to which a run-time name is assigned, including PCBs. Uniqueness here extends to a distinction between a capital letter and its lowercase form. (Because of its intrinsic nature, a logical-name structure must be named.) 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 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 that 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):



MLO–402–87

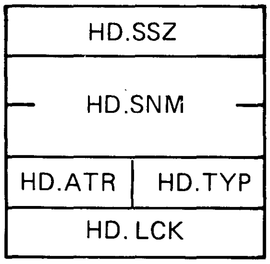
In the previous format:

- The SN.NAM field 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 QUEDF$ system macro as negative offsets from the start of the structure body. (Run-time 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. The format of the structure header is as follows:



MLO–403–87

In the previous format:

- HD.SSZ 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 many primitives for validity checking.

- HD.ATR is the structure attribute bits (defined below).
- HD.LCK is reserved for future use.

The HD.xxx symbols are defined by the QUEDF$ system macro as negative offsets from the start of the structure body. (Run-time 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.

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.SRD	Shared region descriptor
ST.LNM	Logical name value
ST.UDF	Unformatted structure

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

Code	Bit Definition
SA$NAM	For any structure type, structure is named if set, unnamed if not.

SA$RIA	For type ST.RBF, determines the ring buffer input access mode as stream or record:
	SA$RIA = SA$RIS (1) for stream mode
	SA$RIA = SA$RIR (0) for record mode
	Input access mode affects only Conditional Put Element (PELC$) operations.

SA$ROA	For type ST.RBF, determines the ring buffer output access mode as stream or record:
	SA$ROA = SA$ROS (1) for stream mode
	SA$ROA = SA$ROR (0) for record mode
	Output access mode affects only Conditional Get Element (GELC$) operations; stream-mode output access is invalid for Get Element Any (GELA$) operations.

SA$QUO	For types ST.QSM and ST.RBF, determines the packet-queue ordering or the waiting-input-process list ordering, respectively, as by priority or FIFO:
	SA$QUO = SA$IPR (1) for priority ordering
	SA$QUO = SA$IFF (0) for FIFO ordering

Code	Bit Definition

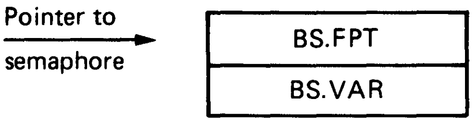
SA$PRO	For types ST.PSM, ST.CSM, and ST.QSM, determines the waiting-input-process list ordering; for type ST.RBF, determines the waiting-output-process list ordering, by priority or FIFO:
	SA$PRO = SA$OPR (1) for priority ordering
	SA$PRO = SA$OFF (0) for FIFO ordering

SA$SRD	For type ST.SRD, determines the shared-region mode as physical or common:
	SA$SRD = SA$PHY (1) for physical mode
	SA$SRD = SA$COM (0) for common mode

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 the value can be decremented.

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



MLO–404–87

In the previous format:

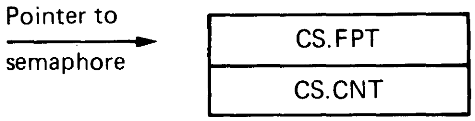
- BS.FPT is the forward pointer to the first waiting process, if any.
- BS.VAR is the semaphore gate variable.

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.

2.2.1.4 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 the variable can be decremented. 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:



MLO–405–87

In the previous format:

- 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.

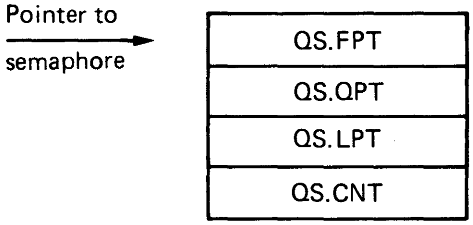
2.2.1.5 Queue Semaphore Definition

A queue semaphore is a further generalization of a counting semaphore. This case has 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. The Put operation 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.

The higher-level Send Data and Receive Data operations are essentially elaborations of the basic Put Packet and Get Packet operations, for use by general or device-access processes in a mapped environment.

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



MLO–406–87

In the previous format:

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

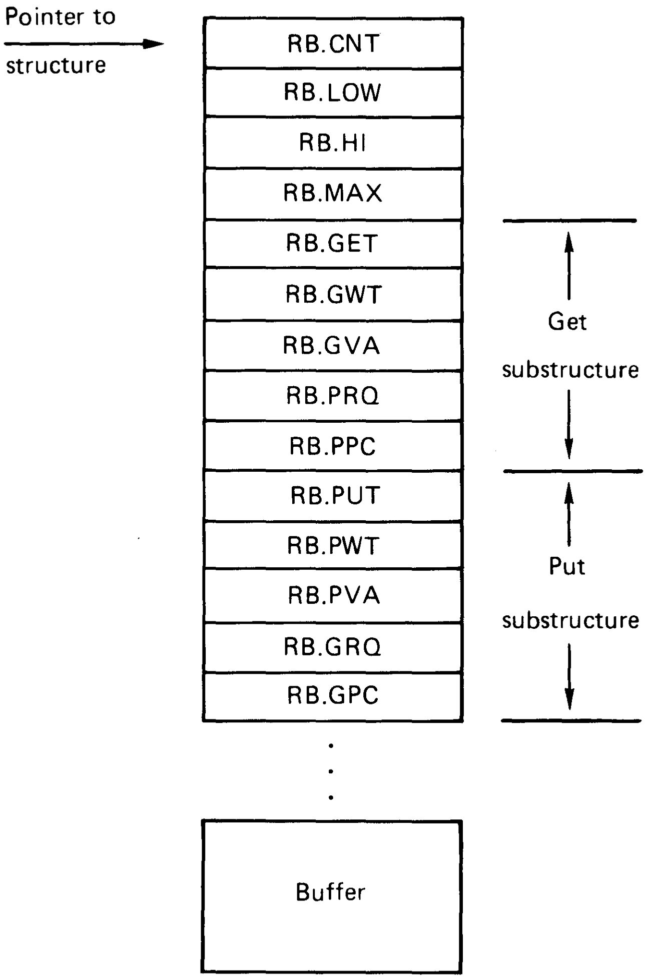
2.2.1.6 Ring Buffer Definition

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 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 GELM$, GELC$, GELA$, PELM$, PELC$, and RBUF$ primitives in Chapter 3 for a complete description of ring buffer operations.

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



MLO-407-87

In the previous format:

- 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—the maximum number of bytes it can contain.
- 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 first process, if any, waiting for active output access—the head of the Get queue (see RB.GPC).
- RB.GVA is the binary gate variable that controls the granting of active output access for Get operations.
- RB.PRQ is a wake-up counter used during concurrent Get/Put operations for awakening the putting process pointed to by RB.PPC.
- RB.PPC is the pointer to the blocked process with active input access to the buffer, if any.
- 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 first process, if any, waiting for active input access—the head of the Put queue (see RB.PPC).
- RB.PVA is the binary gate variable that controls the granting of active input access for Put operations.
- RB.GRQ is a wake-up counter used during concurrent Get/Put operations for awakening the getting process pointed to by RB.GPC.
- RB.GPC is the pointer to the blocked process with active output access to the buffer, if any.

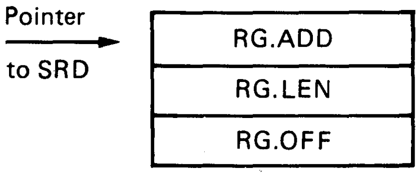
The control structure and the buffer area may not be contiguous in memory.

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 access-mode bit settings affect certain characteristics of the Put (PELM$ and PELC$) and Get (GELM$, GELC$, and GELA$) operations, respectively, as described in Chapter 3.

2.2.1.7 Shared Region Descriptor Definition

A shared region descriptor (SRD) consists of three words that specify the location and extent of a user-defined shareable memory area. The SRD structure allows the indirect association of a structure name to a region of memory. (By the nature of its use, an SRD is normally a named structure.) Other than creation and deletion, the only operation defined on an SRD structure is access shared region, which returns information about the described region. Region-sharing operations are described in Chapter 5.

The format of an SRD, excluding the structure header, is as follows:



MLO–408–87

In the previous format:

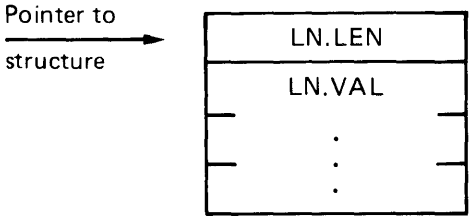
- RG.ADD is a physical memory address, specified as a PAR value in a mapped system or simply as an address in an unmapped system.
- RG.LEN is the length of the region, specified in PAR ticks (units of 32 words) in a mapped system or in bytes in an unmapped system.
- RG.OFF is, for a mapped common region, an offset in bytes from the PAR value (RG.ADD) to the region base.

The SA$SRD bit of the structure-header attribute byte (HD.ATR) indicates the region mode as physical if set or common if clear. Note that the RG.OFF word is not significant for a physical region or an unmapped application.

2.2.1.8 Logical-Name Structure Definition

A logical-name structure contains the translation value, or definition, for a logical name, which is itself a structure name. The structure is variable in size up to 258 bytes and consists of a string-length word followed by a variable-length ASCII character string. Other than creation and deletion, the only explicit operation defined on a logical-name structure is Translate Logical Name, which returns the immediate translation value of a logical name. (Logical-name definitions may be "nested," providing for multiple levels of indirection, since the translation value may represent another structure name.) However, all other primitive operations that operate on typed structures implicitly operate on logical-name structures to obtain the eventual translation of a logical name into another kind of structure name.

The format of a logical-name structure, excluding the mandatory structure name block and the structure header, is as follows:



MLO–409–87

In the previous format:

- LN.LEN is the length in bytes of the character string contained in field LN.VAL; maximum value is 256.
- The LN.VAL field contains a variable-length ASCII character string.

The SA$NAM bit of the structure-header attribute byte (HD.ATR) is always set.

2.2.1.9 Unformatted Structure Definition

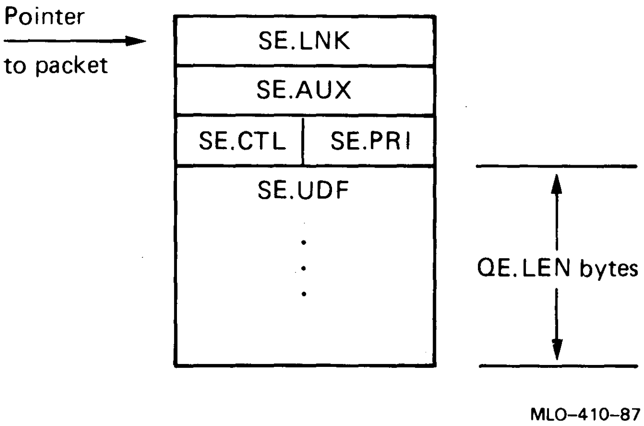
An unformatted structure consists of a data area of user-specified size, preceded by a standard structure header. The kernel does not impose a format on the data area, as no primitives are provided to operate on it. An unformatted structure is allocated by the CRST$ primitive from system-common memory, has the ST.UDF type code, and may be named. Such a structure may be operated on directly by a privileged or driver mapped process or by any process in an unmapped environment. (An unformatted structure might be used in connection with a user-implemented primitive operation, for example.)

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 shareable) kernel resource.

A packet consists of a 3-word packet header and a fixed amount of message space, called the undefined portion. (The header is part of the packet and should not be confused with the prefixed structure header of a typed structure.) The size of a packet is 40 bytes, allowing up to 34 bytes of usable message space, the undefined portion.

The format of a packet is as follows:



In the previous format:

- 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; the subfields of this byte are set by the SEND$ primitive and used by the Receive Data (RCVD$) primitive, as described in Chapter 3.
- 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 in bytes of the undefined portion; the symbol SE.SIZ represents the overall packet size. These symbols, and the SE.xxx offset symbols shown above, are defined by the QUEDF$ system macro.

2.2.3 System Queues

The kernel maintains a number of queues, or linked lists, of dynamically related elements such as PCBs or message packets. Two queuing mechanisms are used: the singly-linked list and the doubly-linked list.

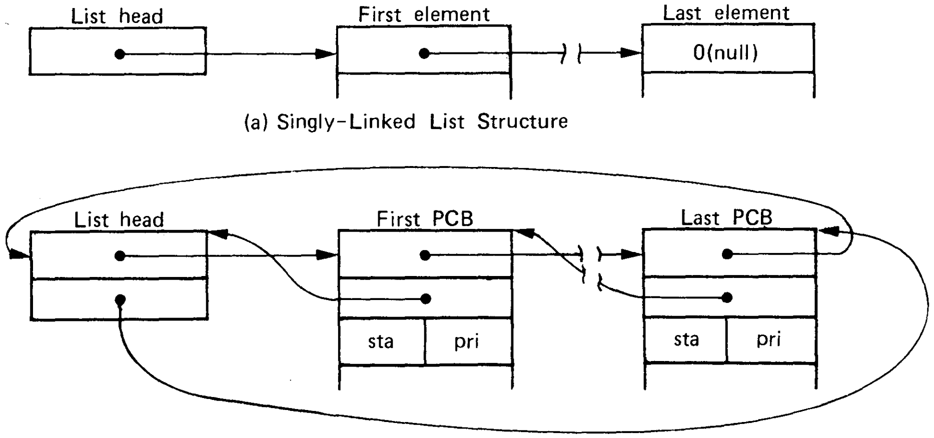
2.2.3.1 Singly-Linked Lists

A singly-linked list structure uses one link word, or pointer, for each list element and is used for the following purposes:

- The blocking queue of a semaphore or a ring buffer (waiting-process list)
- The blocking queue of the kernel's timer-service semaphore (time-ordered list of "sleeping" processes)
- 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 system-common free-memory lists
- The unallocated free-RAM list
- Structure name table lists
- Kernel primitive-resumption list

A singly-linked list is shown schematically in Figure 2–12(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 link word.

Figure 2–12: System Queue Structures



(a) Singly-Linked List Structure

(b) Doubly-Linked List Structure (State Queue)

MLO–445–87

A singly-linked list is normally either FIFO or priority ordered, depending on the ordering attribute associated with the list. (An exception is the blocking queue of the kernel's timer-service semaphore, which has a special time-dependent ordering.) In the case of blocking queues and packet queues, the ordering attribute is determined by the user when the corresponding semaphore or ring buffer is created. (FIFO ordering is the default.) The internal, kernel-maintained lists of free-memory elements are ordered by ascending addresses of the linked elements.

2.2.3.2 Doubly-Linked Lists

A doubly-linked list structure uses two link words (a forward pointer and a backward pointer) for each 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 ready-active state queue is priority ordered; the ready-suspended state queue is LIFO ordered for quick enqueuing. (A doubly-linked list is also used for the inactive queue.)

A doubly-linked list is shown schematically in Figure 2–12(b). The list head (at a fixed location in kernel data space) has the following format, identical to the first three words of a PCB:

SQ.FPT	
SQ.BPT	
SQ.STA	SQ.PRI

MLO–411–87

In the previous format:

- 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 unused.
- SQ.STA is the state code corresponding to the process state represented by the queue—SC.RDA or SC.RDS, for example.

The list is terminated by pointing the forward link of 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 Kernel Data Segment Organization

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

- System-interrupt stack (.10STK p-sect)
- Kernel's private impure data (.20DAT p-sect)
- Interrupt dispatch block area (.30IDB p-sect)

These areas follow the system-common area.

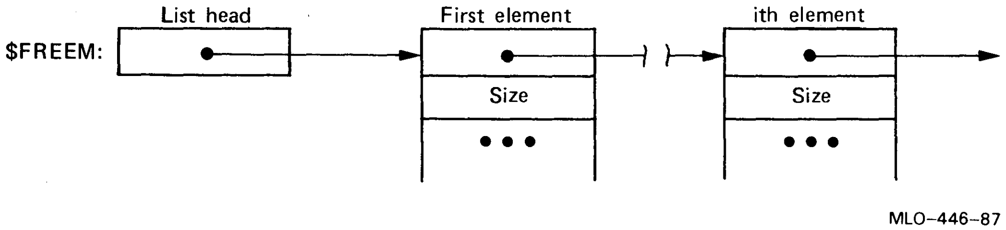
System-common memory is subdivided by the system-initialization (INIT) routine at run time as follows:

- The free-packet pool, from which processes obtain "empty" packets by means of the ALPK$ or ALPC$ primitive. 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 default size of this pool is 800 bytes (20 x 40 bytes); see Section 2.2.2. The available packets are linked into a free-element queue by INIT.
- The free-memory pool, from which all dynamic 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 3000 bytes. After establishing this pool, the INIT routine creates the static-process PCBs in it.

The structure of the free-memory pool is 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 primitive 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 2–13.

Figure 2–13: Free-Memory Pool



A zero pointer value terminates the list. The INIT routine initializes the kernel variable $FREEM 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 beginning 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.

Note

The free-memory pool should not be confused with the free-RAM list, which is described in Chapter 5.

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 used with many primitive service requests. For ease of reference, the primitive descriptions are in alphabetical order, by primitive name.

The MACRO–11 interface consists of a set of keyword macros. The macros facilitate construction of the argument block required by each kernel primitive routine, as well as the invocation of the routine. The three forms of macro call provided for each primitive service permit the following variant usages:

- Run-time construction or modification of the required argument block in user-specified RAM storage
- Run-time construction of the argument block on the user's stack
- 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 and dedicated to that 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
```

R0 must point to the caller's argument block when the IOT is executed. The primitive service macro calls generate this sequence as part of their expansion.

The primitive name (prim in the previous example) 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 $, $S, or $P to the mnemonic (for example, CRST$, CRST$S, or CRST$P). The suffixes $, $S, and $P identify variant forms of the basic macro call; the three variants provide maximum coding flexibility and efficiency. The three variants differ as follows:

- The variant prim$ is used for run-time construction or modification of both the required argument block in a preallocated 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$P macro variant.)
- The variant prim$S is used for dynamic generation of both the required argument block on the user's stack and the IOT sequence. This variant is useful if a static argument block area is not desirable, as for a primitive that is executed only once or infrequently.
- The variant prim$P is used for assembly-time generation, in ROM or pure RAM, of the argument block only. (No IOT sequence is generated.) This variant is used with the null- or single-argument form of the prim$ variant, as described below. The prim$P variant may also be used without arguments for convenient allocation of an area of the correct size for a given argument block in impure RAM, to be used with the "full" form of the prim$ variant.

The radix for any MACRO–11 argument value is octal, unless you put a decimal point after the value.

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,RO
MOV     argument_1,(RO)
MOV     argument_2,2(RO)
 .
 .
 .
MOV     argument_N,N*2(RO)
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
```

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
```

This expansion allows for precall 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, R0 is assumed to be preset to 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:

- 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.
- 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$S

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

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

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
< code for popping 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
```

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:

- If one macro argument is specified, all arguments must be specified, except where a default value is explicitly described for a given argument. The stack is purged of all arguments on return from the call.
- 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
```

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$P macro variant can be used within the scope of a PDAT$ (pure-data p-sect) macro to generate an argument block in ROM or write-protected RAM storage. (This usage implies 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 then be used to pass the address of the block to the appropriate primitive.

Alternatively, the prim$P macro can be used within the scope of an IMPUR$ (impure-data p-sect) macro to allocate an argument block area of the required size in read/write storage. The argument list is not needed for this purpose, since the argument block must be filled in at run time. The argument block is filled in prior to the issuing of a "prim$ area" call.

3.1.4 Error Returns

An error condition encountered by a primitive service routine is reported to the caller by a return of control to the call site with the carry (C) bit set in the processor status word (PSW). An exception code identifying the error condition is returned in R0. Therefore, the caller should test the C bit following a primitive call to detect a possible error return and evaluate the content of R0.

Note

> Some primitives alternatively return a nonerror function value in R0, such as a TRUE or FALSE indication from a conditional primitive operation. In that case, the C bit is clear on return from the primitive, distinguishing the R0 function value from a possible exception code.

Exception codes and types are described in general in Section 6.1 and Table 6–1. Collectively, the primitive routines return only a limited subset of the exception codes of types EX$SVC (system service) and EX$RSC (resource). (The return of some address-check exceptions is conditional on the CHECK option of the SYSTEM configuration macro; see Chapter 4.) That subset of possible exception code values, mapped by globally defined symbols of the form ES$xxx, is as follows:

- Codes for conditions of type EX$SVC

ES$AOV	Already owned vector, cannot connect
ES$IAD	Invalid address: odd or not in user's virtual space
ES$IPM	Illegal parameter
ES$IPR	Illegal primitive for context
ES$IST	Invalid structure descriptor
ES$IVC	Illegal vector address
ES$NID	No interrupt dispatch block established for vector
ES$SIU	Structure is in use
ES$SNI	Structure name already in use

- Codes for conditions of type EX$RSC

ES$NFA	No free APR for window mapping
ES$NMK	Insufficient space for creation of a dynamic kernel structure

The exception symbol values are defined by the EXMSK$ macro in the COMM and COMU system macro libraries. The particular exception codes returned by a given primitive are specified in the description of that primitive.

A process may elect to raise an exception, by means of the Report Exception (REXC$) primitive, based on the exception code it receives as an error return. The REXC$ primitive requires an exception type value as well as an exception code as calling parameters. Before using REXC$, therefore, the reporting process must derive an exception-type mask value from type information that is encoded in every exception code value. Section 6.2.1 shows a MACRO–11 program fragment that performs the required code-to-type transformation.

3.1.5 Structure Descriptor Block (SDB) Usage

A structure descriptor block (SDB) describes a particular kernel data structure, such as a semaphore, ring buffer, or logical-name translation value. (These dynamic typed structures, described in Section 2.2.1, are allocated in kernel space but are created, used, and deleted at user request.) Many primitives act on a given structure; therefore, the structure must be identified (indirectly) in the corresponding primitive request.

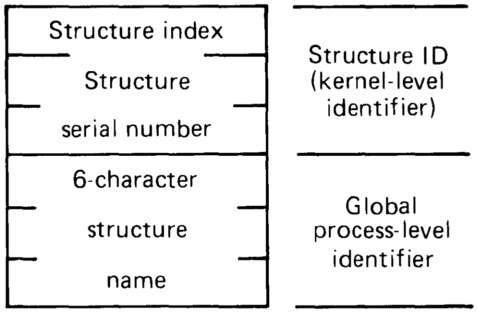
An example of such a request is the Signal Semaphore (SGNL$) primitive call, which has the form:

```
SGNL$ area,sdb
```

The sdb argument, which is a pointer to an 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 later.

The format of an SDB is as follows:



MLO–412–87

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

An SDB has three uses, as follows:

- To specify the name, if any, of a structure to be created by the CRST$ or CRSR$ primitive.
- To specify a logical name, that is, the name of a logical-name value structure to be created by the CRLN$ primitive.
- To access, through other primitive services, an existing structure that is referenced by either structure ID or structure name. (The reference may be indirect, through an intermediate logical name.)

When a structure is either created or accessed by structure name, 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. Such 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 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. (A process name describes a process control block, another kind of system structure.) A logical name is a form of structure name (the name of a data structure of type ST.LNM), which contains the translation value for the name. Therefore, an SDB containing a logical name is subject to the same rules as an SDB for any other kind of named structure. However, since the translation value of a logical name can itself be another structure name, logical-name references are treated differently from other structure references by most primitives, as described in Section 3.1.5.3.

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 SGNL$ or WAIT$ request, for example), you must initialize the SDB as follows:

1. Set the value of the first word (structure index) to 0.
2. Ensure that the structure name field contains 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. Therefore, any unused high-order bytes of the structure name field in the SDB should 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 drivers. (The driver 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 may be created by passing the CRST$ 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. Set the value of the first word (structure index) to 0.
2. Make sure that the value of the seventh byte (first byte of the structure name field) is 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, 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, then, an unnamed structure is used only for internal synchronization or communication between processes in the same static process family; that is, among processes residing in the same address space.

3.1.5.3 Implicit Translation of Logical Names

The primary design intention for logical names is to permit a level of indirection within file specifications that are passed to an I/O ancillary control process (ACP), such as the RTACP. In addition to this and other possible uses, however, a logical name can also be used as an alias for another structure name.

All primitives that operate on existing structures other than PCBs will accept a logical name as a structure reference. The reference can be either by name or by structure ID, as described for named structures in general. Except for the Translate Logical Name (TRLN$) and Delete Logical Name (DLLN$) primitives, none of those primitives operates directly on the identified logical-name structure. Instead, they attempt to translate the logical-name reference into a valid reference to a kind of structure they do operate on, such as a semaphore or a ring buffer.

The rules for such primitive operations with respect to logical names are:

1. If the SDB passed to the primitive identifies a logical-name structure, the primitive obtains the corresponding translation string.
2. If the translation string exceeds six characters and thus is not a structure name, the primitive returns an "invalid structure descriptor" (ES$IST) error.
3. If the translation string does not exceed six characters, the primitive performs a structure lookup, using the translation string as the structure name. (If the translation string is shorter than six characters, the kernel's name-lookup mechanism will pad out the string with trailing NULLs, **not** space characters.)
4. If the lookup produces another logical-name structure, the primitive repeats the translation and attempted lookup procedure according to rules 2, 3, and 4.
5. If the lookup produces an existing structure of a type appropriate to the primitive operation, the primitive performs the requested operation.

   Note

   Following a successful lookup, both the name and the structure ID fields of the SDB are updated to reflect the structure to be operated on rather than the original logical-name structure.

6. If the lookup fails (no matching structure) or finds an existing structure of an invalid type for the primitive operation, the primitive returns an invalid structure descriptor (ES$IST) error.

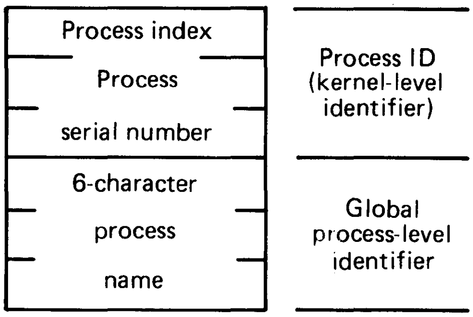
The Create Structure (CRST$) primitive, which creates a semaphore, ring buffer, or unformatted structure, and the Create Shared Region (CRSR$) primitive, which creates a shared region descriptor (SRD) structure, will also accept a logical name as an indirect specification of the name to be given to the created structure. (This use of logical names is less likely than their use for indirect access to an existing structure, however.) The CRST$ and CRSR$ primitives operate with respect to logical names according to the following rules (rules 1 through 4 are the same as those for structure access):

1. If the SDB passed to the primitive identifies a logical-name structure, the primitive obtains the corresponding translation string.
2. If the translation string exceeds six characters and thus does not qualify as a structure name, the primitive returns a "structure name already in use" (ES$SNI) error.
3. If the translation string does not exceed six characters, the primitive performs a structure lookup, using the translation string as the structure name. (If the translation string is shorter than six characters, the kernel's name-lookup mechanism will pad out the string with trailing NULLs, **not** space characters.)
4. If the lookup produces another logical-name structure, the primitive repeats the translation and attempted lookup procedure according to rules 2, 3, and 4.
5. If the lookup produces an existing structure of a type other than logical name, the primitive returns a "structure name already in use" (ES$SNI) error.
6. If the lookup fails to find a matching named structure, the primitive creates the requested structure with the name used in the final lookup operation.

See the CRLN$, TRLN$, and DLLN$ primitives for a description of the specific logical-name operations: creation, single-level translation, and deletion, respectively. Only those primitives create or operate on logical-name structures.

3.1.6 Process Descriptor Block (PDB) Usage

A process descriptor block (PDB) describes a process. The PDB identifies a process control block (PCB), the kernel structure that represents an existing process and contains the state and context information for that process. Although PDBs and SDBs are structurally identical, they differ somewhat in the way they are interpreted and treated by some primitive operations. Like an SDB, 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:



MLO–413–87

The use of a PDB for process creation (CRPC$ primitive) is identical to the use of an SDB for structure creation, as described above. The use of a PDB in requests for a primitive operation on another process is also the same as the use of an SDB for reference to an existing structure.

However, the primitives that operate on existing processes provide a shorthand way for the calling process to identify itself as the process to be acted on rather than another process. The shorthand rule is that, in a process-related request, if the PDB argument value is 0 (implying no PDB) or the content of the specified PDB is null, the request will operate reflexively on the calling process.

Reference to an existing process is best illustrated by example. The following primitives operate on an existing process and, possibly, on the calling process:

- CHGP$ (Change Process Priority)
- GMAP$ (Get Process Mapping)
- GTST$ (Get Process State)
- SPND$ (Suspend Process)
- STPC$ (Stop Process)

Each of those primitives requires the address of a PDB as a calling argument and interprets that argument in a consistent manner. For example, the call for the GTST$ primitive, which returns information about a given process, is of the form:

```
GTST$ area,pdb,buf
```

The pointer to 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 GTST$, 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 (see step 2).
2. If the process index field 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 field is 0, the primitive examines the process name field (see step 3).
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 the process index value is 0, the primitive writes a valid process identifier in the process ID field as an implicit part of the primitive operation. This action is like that performed for a structure access by structure name, as described in Section 3.1.5, and permits optimal access to the process on subsequent uses of the PDB.

3.2 ACSR$ (Access Shared Region)

Pascal equivalent: ACCESS_SHARED_REGION Procedure

The Access Shared Region (ACSR$) primitive lets the calling process gain access to a region of memory that was previously made shareable by another process, by means of a run-time name assigned to the shared region. (The ACSR$ primitive can also be used to access a shared region that was defined at build time by a MEMORY configuration macro.) More precisely, the ACSR$ primitive returns a physical description of the named shared region to a region ID block (RIB) that is pointed to in the call. The RIB information is normally used in a subsequent window-mapping operation, performed for general mapped processes by the MAPW$ primitive.

The accessed region can be either a common or physical shared region (see the CRSR$ primitive). The information returned in the RIB describes the region's location, size, and mode attribute. The location of a shared region is represented by a combination of the region base, specified by a physical PAR value, and the region offset, specified as a displacement in bytes from the base. The region size is described in PAR ticks (32-word units). The size of a common region as described in the RIB can therefore exceed the size declared by the region's creator by up to 31 words. (The creator and accessors should have common size definitions independent of the RIB description where necessary.) Since a physical region is located on a 32-word physical boundary and allocated in 32-word units, its offset is 0, and the described size, in PAR ticks, represents the exact amount of space allocated for the region.

Although region sharing by means of the kernel is applicable primarily to a mapped target environment, the CRSR$/ACSR$ primitives can be useful in an unmapped application containing more than one user static process. (Because of the single address space in an unmapped system, however, having multiple user static processes is generally of no advantage.) Coding details differ for unmapped usage, since there is no distinction between virtual and physical addresses. In the unmapped case, the RIB specifies the base of the region directly as a physical address, and the region size is represented in bytes. The base and size information supplied in the RIB is used directly. Also, common and physical regions are effectively equivalent; the region offset is 0.

A semaphore is usually required to protect against concurrent references to a region shared by several processes. Also, the kernel structure (shared region descriptor) that represents a shared region can be deleted by the DLSR$ primitive, although typically that is done only if the creating static process terminates. The kernel does not provide any automatic safeguard against inadvertent reference to a deleted (and possibly deallocated) shared region, since any process that accessed the region while shareable retains a description of it.

Chapter 5 contains a general discussion of region sharing, including the use of ACSR$ in the context of the related primitives ALRG$, CRSR$, DLRG$, DLSR$, MAPW$, and UMAP$. The CRSR$ primitive provides the complementary Create Shared Region operation, which declares a region as being shareable and assigns its run-time name.

Syntax

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

Variant	Macro Call
ACSR$	ACSR$ [area,sdb,rib]
ACSR$S	ACSR$S [sdb,rib]
ACSR$P	ACSR$P [sdb,rib]

area

The address of a user-memory location at 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 of the shared region to be accessed (that is, the name associated with the corresponding kernel SRD structure) and in which the kernel returns information identifying the SRD. See Section 3.1.5 for the format and use of an SDB. This argument has the form:

```
[SDB=]sdb-address
```

rib

The address of a 4-word (RI.SIZ bytes) area in user memory, the region ID block, in which the location, size, and mode attribute of the allocated region is returned by the primitive, as described under Semantics. This argument has the form:

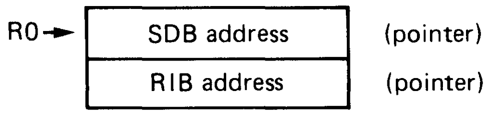
```
[RIB=]area-address
```

Restrictions

This primitive may be used only at process level; it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the ACSR$x macro has the following format:



MLO-414-87

Syntax Example

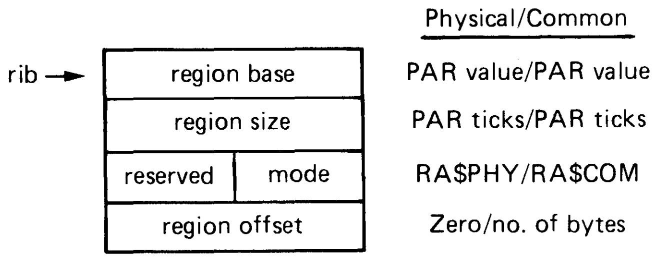
```
ACSR$S sdb=#SRGNAM,rib=#REGDSC
```

This stack ($S) form of the macro call specifies the location of the structure descriptor block SRGNAM containing the name of the region to be accessed and specifies the location of the region ID block REGDSC in which the region description is to be returned. (See the CRSR$ primitive description for a corresponding region-creation example.)

Semantics

The ACSR$ primitive looks for a shared region descriptor (SRD) having the name specified in the caller's SDB. If that SRD exists, the primitive copies information in the SRD to the RIB specified in the call and returns to the caller. If no such SRD exists, the primitive returns to the caller, with an error indication.

Information describing the accessed region is returned in the user's RIB area in the following form, assuming a mapped environment:



MLO-415-87

The offset and size symbols defined for the RIB fields are:

RI.ADD	Region base
RI.LEN	Region size
RI.ATR	Region mode (attribute byte)
RI.RES	Reserved (high byte)
RI.OFF	Region offset
RI.SIZ	RIB size in bytes

The RIBDF$ macro in the MicroPower/Pascal COMU and COMM system macro libraries defines these symbols.

In the mapped environment, the region base is returned as a physical PAR value, representing a 32-word physical boundary. (That value is not directly usable as an address, of course, but can be used in a physical-to-virtual mapping operation as implemented by the MAPW$ primitive.) The region offset, relevant for a shared common region, is an increment in bytes from the PAR value to the beginning of the region. (The region-offset field is significant for the Map Window operation.) The region size specifies the number of PAR ticks (units of 32 words) in the region. In the case of a common region, the described size represents the actual size of the region (specified to CRSR$ in bytes) rounded up to the next multiple of 32 words. The region mode is indicated by the value of the mode symbol RA$PHY or RA$COM, denoting a physical or common region, respectively. (The RA$xxx symbols are defined by the RIBDF$ macro.)

In an unmapped environment, the region base is a physical address that can be used directly, and the region size is the number of bytes specified in the Create Shared Region request. The region offset is always 0, regardless of the region mode.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the RIB address is not on a word boundary.

ES$IST Invalid structure description (index or name); no such shared region descriptor exists. (This error return could be caused by an erroneous SDB address.)

3.3 ALPC$ (Conditionally Allocate Packet)

Pascal equivalent: COND_ALLOCATE_PACKET Function

The Conditionally Allocate Packet (ALPC$) 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 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 a Signal Queue Semaphore (SGLQ$) or Conditionally Signal Queue Semaphore (SGQC$) primitive operation, without blocking if a packet is not available. (Compare with ALPK$, the unconditional form.)

The DAPK$ primitive is the inverse of ALPC$, allowing a process to deallocate a message packet.

Syntax

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

Variant	Macro Call
ALPC$	ALPC$ [area,qelm]
ALPC$S	ALPC$S [qelm]
ALPC$P	ALPC$P

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

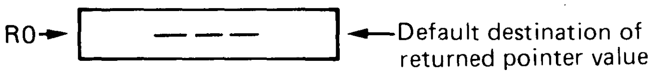
If the qelm argument is null, the packet pointer returned by the primitive is available only in the calling argument block. If the argument is null in the stack ($S) version of the macro call, the returned pointer value is left on the stack. (Ordinarily, the argument block is purged from the stack following the call.) In the parameters-only ($P) 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 following Restrictions section.)

Restrictions

The argument block must be in read/write memory.

Argument Block

The calling argument block generated (or assumed to exist) by the ALPC$x macro has the following format:



MLO–416–87

Semantics

The ALPC$ 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 value TRUE in R0.

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

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. In the current version of MicroPower/Pascal, the values are 1 and 0, respectively.

Error Returns

None.

3.4 ALPK$ (Allocate Packet)

Pascal equivalent: ALLOCATE_PACKET Procedure

The Allocate Packet (ALPK$) 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 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 either the Signal Queue Semaphore (SGLQ$) or the Conditionally Signal Queue Semaphore (SGQC$) 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 ALPK$ is the DAPK$ primitive, allowing a process to deallocate a message packet.

Syntax

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

Variant	Macro Call
ALPK$	ALPK$ [area,qelm]
ALPK$S	ALPK$S [qelm]
ALPK$P	ALPK$P

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, the address may be null. If specified, it must be a word address. Because the argument you specify is expanded directly into an MOV instruction destination argument, it should not contain an immediate expression indicator (#).

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 ($S) version of the macro call, the returned pointer value is left on the stack. (Ordinarily, the argument block is purged from the stack following the call.) In the parameters-only ($P) 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 following Restrictions section.)

Restrictions

The argument block must be in read/write memory.

Argument Block

The calling argument block generated (or assumed to exist) by the ALPK$x macro has the following format:

R0
— — —
Default destination of
returned pointer value

MLO–417–87

Semantics

The ALPK$ 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 no packets are free, 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 DAPK$ primitive.)

Error Returns

None.

3.5 ALRG$ (Allocate Region)

Pascal equivalent: ALLOCATE_REGION Function

The Allocate Region (ALRG$) primitive allocates an area of unused physical memory, if available, to the calling process. The memory area, called a region, is of user-specified size and is allocated dynamically from a list of free-RAM segments maintained by the kernel. (See Sections 5.1 and 5.3.) If a region is successfully allocated, the primitive returns control to the calling process, with a Boolean TRUE value in R0 (R0=1), and other information as described below. If a region of the required size cannot be allocated, the primitive returns control to the caller, with a Boolean FALSE value in R0 (R0=0).

Allocation is achieved through a user-supplied region ID block (RIB), in which the primitive returns information about the location and size of the allocated region. The process that "owns" the RIB is completely responsible for the region and can use it for any purpose; the kernel does not keep track of the allocated space. A physical region can be deallocated by the DLRG$ primitive when the space is no longer needed.

Although dynamic RAM allocation is designed primarily for a mapped target environment, the ALRG$ primitive can be used in an unmapped application as well. Coding details differ between mapped and unmapped usage. In the mapped case, the caller specifies the required region size in term of PAR ticks; units of 32-word blocks (100 octal bytes). The primitive returns the physical base address of the region as a page address register (PAR) value and returns the region size in PAR ticks. This PAR information, returned in the RIB, can be used in subsequent window-mapping operations, implemented by the MAPW$ primitive.

In the unmapped case, the caller specifies the required region size directly in bytes. The primitive returns the base address of the region directly, of course, and returns the region size in bytes, rounded up to the next multiple of 4, if necessary.

Chapter 5 contains a general discussion of dynamic RAM allocation, including the use of ALRG$ in the context of the related primitives ACSR$, CRSR$, DLRG$, DLSR$, MAPW$, and UMAP$.

Syntax

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

Variant	Macro Call
ALRG$	ALRG$ [area,size,rib]
ALRG$S	ALRG$S [size,rib]
ALRG$P	ALRG$P [size,rib]

area

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

```
[AREA=]arg-blk-address
```

size

A value that specifies the size of the region to be allocated. For a mapped application, the size value specifies the number of 32-word (64-byte) blocks required. For an unmapped application, the size value specifies the number of bytes required. This argument has the form:

```
[SIZE=]integer
```

rib

The address of a 4-word (RI.SIZ bytes) area in user memory, the region ID block, in which the location, size, and mode attribute of the allocated region is returned by the primitive, as described under Semantics. (The mode of a dynamically allocated region is always physical.) This argument has the form:

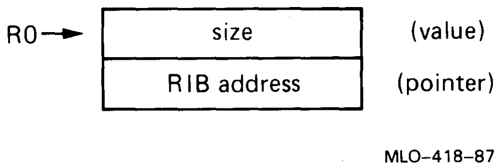
```
[RIB=]area-address
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the ALRG$x macro has the following format:



Syntax Example

```
ALRG$ area=#ALARGS,size=#200,rib=#8KBREG
```

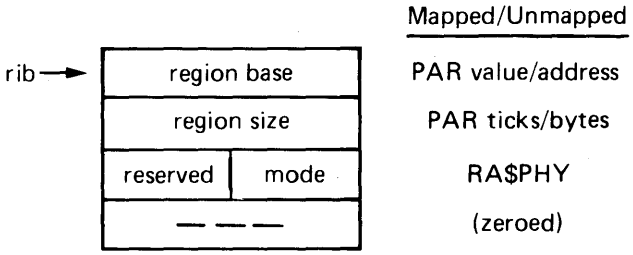
Assuming a mapped target system, this macro call requests an 8192-byte region, specified in octal as 200 PAR ticks, that is, 200 units of 100(octal) bytes, each (20000/100). (A physical region of that size can be mapped exactly by one PAR.) The 4-word user area located at 8KBREG will receive the information returned by the primitive describing the allocated region.

Semantics

The ALRG$ primitive checks the kernel's free-RAM list for a memory segment that equals or exceeds the size of the requested region. If such a segment exists, the primitive removes the required amount of memory from the free-RAM list, modifies the caller's RIB area as described below, sets R0 to 1, and returns control to the caller. (The primitive allocates from the free-RAM list on a first-fit basis.) If no sufficiently large free-RAM segment exists, the primitive clears the caller's R0 and returns control to the calling process.

In either case, the user's C bit is clear, distinguishing the value returned in R0 from an error-return indication.

When a region is allocated, the following information is returned in the user's RIB area:



MLO-419-87

The offset and size symbols defined for the RIB fields are:

RI.ADD	Region base
RI.LEN	Region size
RI.ATR	Region mode (attribute byte)
RI.RES	Reserved (high byte)
RI.OFF	Region offset
RI.SIZ	RIB size in bytes

The RIBDF$ macro in the MicroPower/Pascal COMU and COMM system macro libraries defines these symbols.

In a mapped environment, the region base, always on a 32-word physical boundary, is returned as a physical PAR value. (That value is not directly usable as an address, of course, but can be used in a physical-to-virtual mapping operation as provided by the MAPW$ primitive.) The region size is an integer representing the number of PAR ticks (100 octal bytes) allocated, as represented in the allocation request.

In an unmapped environment, the region base is a physical address that can be used directly, and the region size is an integer representing the number of bytes allocated. If the requested number of bytes was not a multiple of 4, the next higher multiple of four bytes is allocated.

In both cases, the region mode is indicated by the value of the symbol RA$PHY, denoting a physical region. (The RA$xxx mode symbols are defined by the RIBDF$ macro.) The mode of a region (physical or common) is significant to the Create Shared Region (CRSR$) primitive and, indirectly, to the Map Window (MAPW$) primitive. The last word of the RIB, the region-offset field, is not relevant for region allocation; the field is significant only in operations on shared common regions. The ALRG$ primitive sets the word to 0 as appropriate for a physical region.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IAD Invalid address; the RIB address is not on a word boundary.

3.6 CCND$ (Connect to Exception Condition)

Pascal equivalent: $\left\{\begin{array}{l}\text{CONNECT\_EXCEPTION Procedure}\\ \text{DISCONNECT\_EXCEPTION Procedure}\end{array}\right\}$

The Connect to Exception Condition (CCND$) primitive establishes a process as the exception handler for a particular group of processes and for a specified type of exception. (See Chapter 6 for a general discussion of exception handling.) The primitive establishes an existing queue semaphore, identified 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 WAIQ$ operation on its exception queue semaphore.

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

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 through use of the Dismiss Exception Condition (DEXC$) primitive. (See also the SERA$ primitive for exception servicing within the faulting process.)

Syntax

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

Variant	Macro Call
CCND$	CCND$ [area,mask,group,sdb]
CCND$S	CCND$S [mask,group,sdb]
CCND$P	CCND$P [mask,group,sdb]

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 a predefined bit-mask symbol, for which the handler is to be established by the current call. The exception type symbols, of the form EX$xxx, are defined by the EXMSK$ macro and are described in Chapter 6. This argument has the form:

```
[MASK=]symbol
```

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 grp argument of the CRPC$ and DFSPC$ 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:

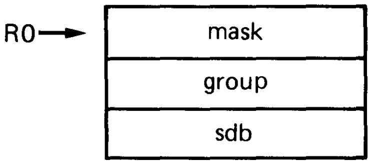
```
[SDB=]sdb-address
```

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 type and group is disconnected from that particular type/group combination.

Argument Block

The calling argument block generated (or assumed to exist) by the CCND$x macro has the following format:



MLO–420–87

Semantics

If a queue semaphore is identified in the call, the CCND$ primitive makes an entry in the kernel's exception-dispatching table to associate the queue semaphore with the specified combination of exception type and exception group(s). The primitive then returns to the caller.

If no queue semaphore is identified in the call (sdb argument value 0), the CCND$ primitive deletes the entry, if any, in the kernel's exception-dispatching table for the specified combination of exception type and exception group(s).

Each exception-dispatching table entry describes one exception type, one exception group code, and the associated queue semaphore. No more than one entry for any one type/group combination is allowed, precluding multiple handlers for a given type and group. Also, if a table entry for a given exception type specifies the wildcard group code (0), no other entry may exist for the same exception type. Otherwise, many table entries may exist for each exception type. Chapter 6 describes the kernel's exception-dispatching mechanism.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; either no bits were set in the mask word or more than one bit was set.

ES$NMK Resource not available; either the kernel's free-memory pool was exhausted (a table entry could not be allocated for the connection) or an entry exists for the specified type/group combination.

Implementation Notes

The group code permits several exception handlers for the same exception condition to coexist, each handler implementing a management strategy suited to one or more groups of processes. If one exception-management strategy is applicable to several groups for an exception type, several CCND$ calls can be used to connect one type of exception from several exception groups to the same exception queue. Alternatively, several CCND$ calls can be used to connect several types of exceptions from one exception group to the same exception queue.

Care should be taken in the use of the wildcard group code, 0, which implies all exception groups. Although exception handlers in general should not cause exceptions to occur, it is particularly important that a wildcard-group handler not do so, since the wildcard group necessarily includes the handler itself. (Like any other process, a handler must be a member of an exception group.) If any handler (wildcard group or otherwise) causes an exception of a type handled by itself, the handler will lock up indefinitely in the exception-wait state, as will any other process that subsequently causes an exception of the same type.

3.7 CHGP$ (Change Process Priority)

Pascal equivalent: CHANGE_PRIORITY Procedure

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

Typically, this primitive lets a process lower its priority to a normal operating level (less than 128 for a noncritical process) after starting at a high priority level for initialization purposes. The special start-up priorities for static processes are 248 to 255, as described in Appendix A. The highest start-up priority, 255, is used by the most critical static process in an application (for example, an error logger) to execute a 1-time initialization sequence involving the creation of globally needed data structures. 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, again for initialization purposes.

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 running priority, which may, in fact, be higher than the running priority 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.

Note

The functionality of CHGP$ has been extended for MicroPower/Pascal Version 2.0, with corresponding changes to both the macro call and the calling argument block. However, the older form of the macro call, with no pdb argument, will assemble correctly with a Version 2.0 or later COMx macro library.

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,pdb]
CHGP$S	CHGP$S [pri,pdb]
CHGP$P	CHGP$P [pri,pdb]

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 subject process. This argument has the form:

```
[PRI=]priority-value
```

The value must be from 0 to 255.

pdb

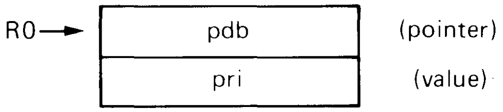
The address of the process descriptor block (PDB) that identifies the process to be acted on or 0. If #0 is specified or the argument is null, the calling process is implied. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:

```
[PDB=]pdb-address or #0
```

The argument default value is 0 in all forms of the macro.

Argument Block

The calling argument block generated (or assumed to exist) by the CHGP$x macro has the following format:



MLO-421-87

Note that the macro-call argument order is reversed in the argument block.

Syntax Example

```
CHGP$S pri=#125.
```

This call sets the calling process's priority to 125(decimal).

Semantics

The CHGP$ primitive places the specified priority value in the PC.PRI field of the specified or implied PCB and calls the scheduler. Thus, the calling process will be preempted if any process in the ready-active queue has a higher priority than the caller as a result of the call. Otherwise, control returns to the calling process.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IPM Illegal parameter; the specified priority value was not within the range 0 to 255.

ES$IST Invalid structure descriptor (index or name); no such process exists. (This error return could be caused by an invalid PDB address.)

3.8 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 driver and to define the ISR code segment. Chapter 7 provides a general discussion of interrupt dispatching and the coding of ISRs. In a mapped environment, the CINT$ primitive is normally used only by a process with the PT.DRV (driver) mapping type.

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]

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 CPU priority level at which the ISR is to execute when entered. If priority-level 7 is requested (that is, PS = 340) a special form of ISR dispatching is implied (see Chapter 7). Note that the CC bits can also be set with this argument, but the T bit cannot. 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 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 address to be passed to the ISR in R3. In mapped systems, if the PIC argument (see below) is TRUE, the value is assumed to be the starting address of the ISR's impure area, which is adjusted as necessary to fall in the range of APR 3 virtual addresses and is passed to the ISR in R3 on interrupt. (The ISR's kernel-mode APR 3 is remapped accordingly when the interrupt occurs.) If the PIC argument is FALSE, the address value is checked to ensure that it is already in the APR 3 range and is passed unchanged to the ISR in R3. (The process's user-mode APR 3 mapping is used to remap kernel-mode APR 3 on interrupt in that case; see Restrictions.) This parameter is typically used to pass the base address of the ISR's impure area. This argument has the form:

```
[IMP=]impure-area-address
```

isr

The address of the ISR code segment. In mapped systems, if the PIC argument (see below) is TRUE, the value is used to determine the proper mapping of the ISR's kernel-mode APR 2 when an interrupt occurs. If the PIC argument is FALSE, the address value is checked to make sure that it is already in the APR 2 range, and the process's user-mode APR 2 mapping is used "as is" to remap kernel-mode APR 2 on interrupt; see Restrictions. This argument has the form:

```
[ISR=]isr-address
```

pic

A Boolean value indicating that the ISR is implemented in non-PIC code (FALSE) or in PIC code (TRUE). (PIC stands for position-independent code.) This argument has the form:

```
[PIC=]#TRUE or #FALSE
```

The TRUE and FALSE symbol values, defined by the EXMSK$ macro, are currently 1 and 0, respectively. This argument is ignored in an unmapped system. (A PIC-coded ISR is typically used only by a process that performs a CINT$ but does not have driver mapping. The Pascal OTS uses the PIC option to implement the CONNECT_SEMAPHORE procedure, which can be used in a process of any mapping type.)

Restrictions

The ISR code segment must not exceed 8128 bytes.

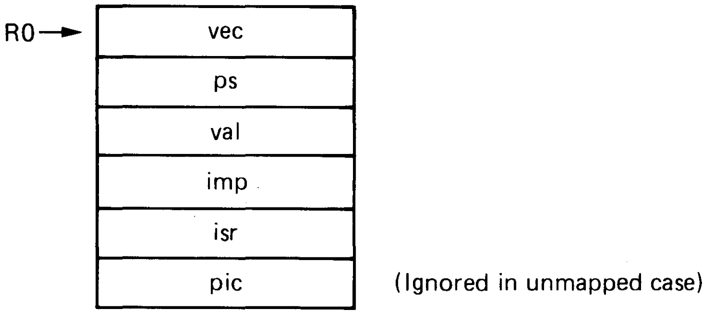
The ISR's impure-data segment must not exceed 8128 bytes.

In a mapped environment, if PIC coding is not used, the combined process/ISR code- and data-segment virtual addresses must be relocated at build time to fall exclusively within the PAR 2 and PAR 3 address ranges, respectively. See the description of driver process mapping in Section 2.1.7.

A module that has a CINT$ primitive should not be added to a supervisor-mode library.

Argument Block

The calling argument block generated (or assumed to exist) by the CINT$x macro has the following format:



MLO-422-87

Syntax Example

```
CINT$ area=#CAREA,vec=#300,ps=#200,val=#0,imp=#DATA,isr=#DEVISR,pic=#FALSE
```

Semantics

The CINT$ primitive sets up the interrupt dispatch block (IDB) associated with the specified vector, causing interrupts through that vector 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 7 contains information closely related to the use of CINT$ and the coding of ISRs. That chapter 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

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$AOV Already owned vector; the specified vector is already connected.

ES$IAD Invalid address; invalid ISR mapping (mapped systems only).

ES$IVC Illegal vector; the specified vector address is less than 60(octal) or beyond the valid range of vectors established at build time (PROCESSOR macro).

ES$NID No interrupt dispatch block established for vector; the vector address was not specified in the DEVICES macro of the system configuration file.

3.9 CRLN$ (Create Logical Name)

Pascal equivalent: CREATE_LOGICAL_NAME Procedure

The Create Logical Name (CRLN$) primitive allows the caller to define or redefine a 1- to 6-character logical name. More precisely, the CRLN$ primitive creates a kernel data structure containing a user-specified translation string value for a given name. Subsequent instances of the logical name will be automatically translated to the corresponding value by other primitive services that operate on dynamic data structures, as described in Section 3.1.5.3.

The caller supplies the logical name in a structure descriptor block (SDB) and specifies a buffer area that contains the translation-string value. The translation string may be up to 256 characters in length and may contain any ASCII character. An override option permits a preexisting logical-name definition to be replaced, thus redefining the name.

The complementary Translate Logical Name (TRLN$) primitive returns the translation-string value directly associated with a logical name, and the Delete Logical Name (DLLN$) primitive eliminates the translation-string value associated with a currently defined logical name. Logical names may also be defined at build time, with the LOGICAL configuration macro described in Chapter 4.

Syntax

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

Variant	Macro Call
CRLN$	CRLN$ [area,sdb,string,length,opt]
CRLN$S	CRLN$S [sdb,string,length,opt]
CRLN$P	CRLN$P [sdb,string,length,opt]

area

The address of a user-memory location at 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 user-constructed structure descriptor block (SDB) containing the structure name to be defined as a logical name. (See Section 3.1.5 for the format and use of an SDB.) A structure name is mandatory for the CRLN$ primitive. This argument has the form:

```
[SDB=]sdb-address
```

string

The address of a user-memory area that contains the ASCII character string to be used as the translation value for the logical name. (The effective size of the area is determined by the length parameter.) This argument has the form:

```
[STRING=]area-address
```

length

An integer that specifies the length in bytes of the character string beginning at the location pointed to by the string argument. The valid range of the length parameter is 1 to 256(decimal). This argument has the form:

```
[LENGTH=]integer
```

opt

An optional bit symbol, LN$OVR, indicating that the logical name may already exist as such and, if so, that the supplied translation value is to replace the translation value currently associated with the name. This argument has the form:

```
[OPT=]LN$OVR or #0
```

If the option value is 0 or the argument is null, a preexisting logical-name definition will not be overridden, and the primitive will return an error if any definition of the name already exists.

Restrictions

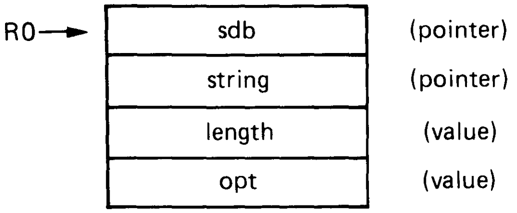
The index field (first word) of the SDB must be zeroed unless the corresponding logical name already exists and the LN$OVR option is specified.

Like other kinds of structure names, a logical name must be unique across all types of kernel data structures.

By system convention, if the translation value of a given logical name is itself intended as a logical name (through serial definitions) and the translation value consists of fewer than six printing characters, the name should be padded to six characters with trailing ASCII spaces in the supplied translation string.

Argument Block

The calling argument block generated (or assumed to exist) by the CRLN$x macro has the following format:



MLO–423–87

Syntax Example

```
CRLN$ area=#LNARGS,sdb=#LGNAME,string=#TRANS,length=#6
```

In this call, the final argument (opt) is null, implying no override if the logical name supplied in the structure descriptor block LGNAME is already defined.

Semantics

The CRLN$ primitive attempts to create a named kernel data structure of type ST.LNM (logical name) large enough to contain the supplied translation string. If the creation is successful, the primitive copies the translation string into the named structure and returns to the caller.

If the specified structure name is already defined as a logical name and the override (LN$OVR) option was specified, the primitive deletes the existing logical-name structure and attempts to create and fill in a new one. If the LN$OVR option was not specified or the structure name is in use as other than a logical, the primitive returns to the caller, with a "name already in use" error indication.

If the structure creation fails for another reason, the primitive returns to the caller with an appropriate error indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IPM Illegal parameter; the specified string length exceeds 256.

ES$NMK Insufficient space for kernel structure; the required logical-name structure could not be allocated.

ES$SNI Structure name in use; the name to be defined as a logical conflicts with an existing structure name.

3.10 CRPC$ (Create Process)

Pascal equivalent: Process-invocation statement

The Create Process (CRPC$) 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 has a combination of the process attributes specified in the service request (for example, priority and exception group) and attributes inherited from the parent process (address space, mapping type, and some context-switch options).

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

The Create Process service is transparent to the Pascal user; no predefined MicroPower/Pascal procedure is equivalent to the CRPC$ 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 CRPC$ macro and their respective macro calls are listed below. (The differences are described in Section 3.1.)

Variant	Macro Call
CRPC$	CRPC$ [area,pdb,pri,cxo,grp,ter,cxl,sti,stl,sth,start,ini]
CRPC$S	CRPC$S [pdb,pri,cxo,grp,ter,cxl,sti,stl,sth,start,ini]
CRPC$P	CRPC$P [pdb,pri,cxo,grp,ter,cxl,sti,stl,sth,start,ini]

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
```

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$FPP	FP–11 floating-point registers
CX$KT	MMU registers (optionally saved, always restored)
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 CXODF$ macro. This argument has the form:

```
[CXO=]option[!option]
```

grp

An integer code value of 1 to 255 indicating the exception-handling group to which the process belongs. The exception-group code value is significant only if one or more exception-handling processes are implemented in the application. Appropriate values are established by design convention. (See the CCND$ primitive and Chapter 6.) This argument has the form:

```
[GRP=]integer-value
```

ter

The entry point of the termination routine 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 value for the process's stack pointer (SP) register. Normally, this value will be the same as the sth argument value, assuming that the first-executed instruction affecting the stack is a push (autodecrement of SP). This argument has the form:

```
[STI=]first-top-of-stack-address
```

stl

The address of the low boundary of the user-allocated process stack, reserved for stack overflow checking. This argument has the form:

```
[STL=]low-bound-address
```

sth

The address of the high boundary of the user-allocated process stack, reserved for stack underflow checking. This argument has the form:

```
[STH=]high-bound-address
```

start

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

```
[START=]first-instruction-address
```

ini

The initial value for location cxl; that is, the value to be stored in that location by the kernel when the process is first executed. This argument is meaningful only if CX$MCX is specified in cxo; otherwise, the value must be 0. This argument has the form:

```
[INI=]word-value or #0
```

Restrictions

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

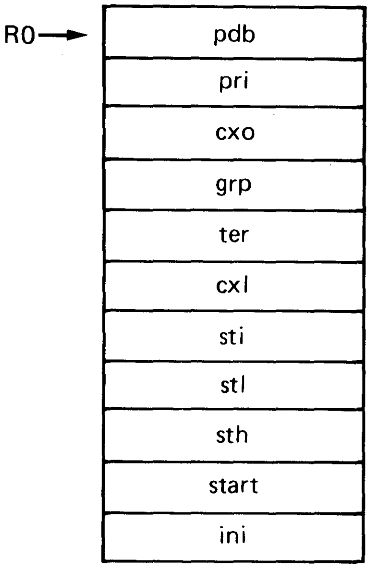
The stack addresses sti, stl, and sth must be word addresses (even values).

The usable area of the process stack lies between stl and sth, exclusively. That is, the value of the user's SP register may range from sth (empty stack) to stl+2 (full stack). The kernel uses the stl and sth locations for dynamic stack-checking purposes, and those locations must not be modified by the user code. (See the Semantics section.)

The size of the process stack in bytes, excluding locations stl and sti, must equal or exceed the value $MINST, which defines the maximum number of bytes that the kernel and ISRs may push on the process stack. In unmapped systems, the value of $MINST is 54(decimal) bytes; in mapped systems, the value of $MINST is 0. When calculating the required stack space for a process, you should add the process's own stack requirement to $MINST.

Argument Block

The calling argument block generated (or assumed to exist) by the CRPC$x macro has the following format:



MLO-424-87

Syntax Example

```
CRPC$S pdb=#P1,pri=#25.,cxo=#CX$FPP,grp=#6,ter=#END,cxl=#0,
       sti=#HIS,stl=#LOS,sth=#HIS,start=#BEGIN,ini=#0
```

Semantics

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

- The attributes and values specified in the call
- The name, if any, contained in the PDB
- Attributes inherited from the parent process (address space and mapping type in a mapped environment) and certain context-switch options unless overridden in the call

The primitive then starts the new process by placing its PCB in the kernel's ready-active queue and calling the scheduler. Thus, the calling process will be preempted if the new process has a higher priority than the caller.

On either immediate or eventual 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 implications of the CRPC$ parameters that are not covered in Section 2.1 are described in the following paragraphs.

The termination entry point (ter) is the location to which control is transferred by the kernel in the event of an exception 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 CX$FPP option (cxo argument) allows a process using FP–11 floating-point instructions to have the contents of the floating-point processor registers saved and restored when it is context switched. (If the option is specified either in a target environment that does not support the FP–11 instruction set or for a process that does not use those instructions, the PCB will be larger than necessary in either case, and needless overhead will be incurred in the latter case.)

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

The CX$KT option causes the mapping registers to be saved during context switch-outs, allowing a process with privileged, driver, or device-access mapping to modify its mapping through direct I/O page access. This option is meaningless in an unmapped system; it incurs needless overhead if, in a mapped environment, it is applied to a process that does not modify its mapping or does so by means of the MAPW$ and UMAP$ primitives.

The kernel places guard words (special values) in the stl and sth locations and tests those guard words during context switch-ins. Modification of either the lower or the upper boundary location will cause a range exception of type EX$RAN, code ES$STO or ES$STU.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; one of the specified address arguments is not on a word boundary or is not in the appropriate address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$NMK Insufficient space for kernel structure; could not allocate the required PCB.

ES$SNI Structure name already in use; a kernel structure already exists with the specified name.

3.11 CRSR$ (Create Shared Region)

Pascal equivalent: CREATE_SHARED_REGION Procedure

The Create Shared Region (CRSR$) primitive allows the calling process to declare a region of memory to be shareable by other static processes and to assign a systemwide run-time name to the region. More precisely, the CRSR$ primitive creates a named kernel data structure, called a shared region descriptor (SRD), that describes the memory region specified by the caller. Subsequently, other processes can gain access to the shared region through the ACSR$ primitive, by means of the run-time name associated with the SRD.

Note

> The CRSR$ primitive is relevant primarily to a mapped memory environment and is described in terms of a mapped application except where indicated otherwise.

A shared region can be either a shared common region or a shared physical region. A common region is one that exists within the caller's statically allocated address space; the location of a shared common region is therefore completely determined by the process declaring it as shared. A physical region is one that was dynamically allocated from unused physical memory by an Allocate Region (ALRG$) operation. Thus, the location of a shared physical region is initially determined by the ALRG$ primitive. See Chapter 5 for a general discussion of common versus physical regions.

Whether common or physical, the region to be made shareable is identified by a region ID block (RIB) in user space that is pointed to in the call. The RIB specifies the region's location, size, and mode attribute. The location, or base, of a common region is specified as a virtual address, the size is specified in bytes, and the mode attribute is "common" (RA$COM). The information describing a common region is placed in the RIB by the user process. (The primitive modifies the information supplied in the caller's RIB for a common region, replacing the virtual description with a physical description, as described under Semantics.)

The base of a physical region is specified as a physical PAR value, the size is specified in PAR ticks (32-word units), and the mode attribute is "physical" (RA$PHY). Normally, the information in the RIB for a physical region is precisely that returned by the prior ALRG$ call that allocated the region.

Although region sharing through the kernel applies primarily to a mapped target environment, the CRSR$ primitive can be useful in an unmapped application containing more than one user static process. (Because of the single address space in an unmapped system, however, generally no advantage is gained from having multiple user static processes.) Coding details differ for unmapped usage, since there is no distinction between virtual and physical addresses. The RIB always specifies the base of a region directly as a physical address, and the region size is represented in bytes. Therefore, the distinction between common and physical regions is not significant for unmapped shared-region creation, although the RA$COM and RA$PHY mode attributes are recognized and applied to the SRD and should be used consistently.

Chapter 5 contains a general discussion of dynamic RAM allocation and region sharing, including the use of CRSR$ in the context of the related primitives ACSR$, ALRG$, DLRG$, DLSR$, MAPW$, and UMAP$. The ACSR$ primitive provides the complementary Access Shared Region operation, which returns RIB information based on a specified shared region name.

Syntax

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

Variant	Macro Call
CRSR$	CRSR$ [area,sdb,rib]
CRSR$S	CRSR$S [sdb,rib]
CRSR$P	CRSR$P [sdb,rib]

area

The address of a user-memory location at 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 of the shared region to be created (that is, the name to be associated with the corresponding kernel SRD structure) and in which the kernel returns information identifying the SRD. See Section 3.1.5 for the format and use of an SDB. This argument has the form:

```
[SDB=]sdb-address
```

rib

The address of a 4-word (RI.SIZ bytes) area in user memory, the region ID block, containing the location, size, and mode attribute of the region to be made shareable, as described under Semantics. In a mapped environment, the region base, size, and offset fields of a RIB for a common region are modified by the primitive; that is, the RIB is both a source and destination parameter in the mapped common case. This argument has the form:

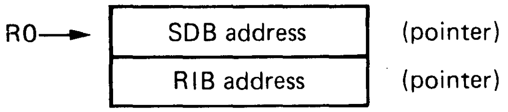
```
[RIB=]area-address
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the CRSR$x macro has the following format:



MLO-447-87

Syntax Example

```
SRARGS:    CRSR$P sdb=SRG8KB,rib=8KBREG
```

This assembly-time, parameters-only ($P) form of the macro call constructs a calling argument block for run-time reference. The call sets up, at location SRARGS, a pointer to the structure descriptor block SRG8KB containing the name to be assigned to the shared region and a pointer to the region ID block 8KBREG that will describe the region to be made shareable. The argument block, in read-only memory, can be used in a run-time call of the form:

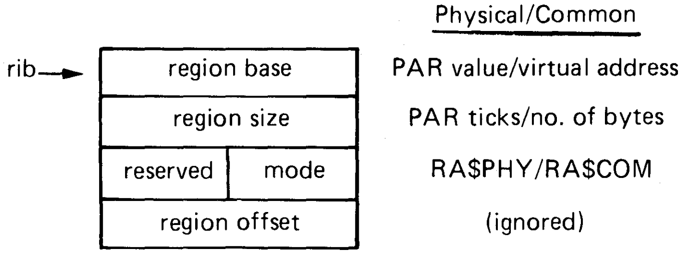
```
CRSR$ SRARGS
```

Semantics

The CRSR$ primitive creates a shared region descriptor (SRD) in the kernel's system-common area, using the region base and size information specified in the caller's RIB. The primitive associates the name specified in the caller's SDB with the SRD structure and returns the structure index and serial number in the SDB. In a mapped environment, if the region mode indicated in the RIB is common (RA$COM), the virtual base and size values supplied in the RIB are converted to a "nearest" physical PAR value and a number of PAR ticks, respectively. The primitive also generates a region-offset value indicating the positive displacement, if any, of the common region base from the calculated PAR value.

For a physical region, no transformation of information between the RIB and the SRD is required, since the values in the RIB are already in the appropriate form.

The information in the user's RIB area prior to the call must be in the following form, assuming a mapped environment:



MLO-425-87

The offset and size symbols defined for the RIB fields are:

RI.ADD	Region base
RI.LEN	Region size
RI.ATR	Region mode (attribute byte)
RI.RES	Reserved (high byte)
RI.OFF	Region offset
RI.SIZ	RIB size in bytes

The RIBDF$ macro in the MicroPower/Pascal COMU and COMM system macro libraries defines these symbols.

On return from the primitive, the RIB for a physical region is unmodified, and the RIB for a common region contains the converted base and size values and the generated offset value already described.

In an unmapped environment, the region base is a physical address, and the region size is an integer representing a number of bytes, as for a mapped common region. The region mode may be either RA$PHY or RA$COM, denoting a physical or common region, although no effective distinction exists between the two in the unmapped case. (The values of the RA$xxx mode symbols are defined by the RIBDF$ macro.)

The last word of the RIB, the region-offset field, is not relevant as input to CRSR$. The field is significant only in shared-common-region mapping operations. Effectively, the offset field value is assumed to be 0 for all operations on mapped physical regions as well as for all unmapped operations.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the RIB address is not on a word boundary.

ES$NMK Insufficient space for kernel structure; the SRD could not be created.

ES$SNI Structure name in use; a kernel structure already exists with the name specified for the region/SRD. (This error return could be caused by an invalid SDB address.)

Implementation Notes

After modification by CRSR$, the caller's RIB for a shared common region contains exactly the same information that would be returned by the Access Shared Region (ACSR$) primitive. Thus, if any process in the same static process family as the creator needed to map a window to the region, the creator's RIB could be used as input to MAPW$ without the need for a call to ACSR$. For example, if the mapping of a sibling dynamic process has diverged from that of the creating process with respect to the region (but not with respect to the RIB) the same RIB can be used by the sibling for remapping.

3.12 CRST$ (Create Structure)

Pascal equivalents: { CREATE_BINARY_SEMAPHORE Function
CREATE_COUNTING_SEMAPHORE Function
CREATE_QUEUE_SEMAPHORE Function
CREATE_RING_BUFFER Function }

The Create Structure (CRST$) primitive creates a semaphore, ring buffer, or unformatted structure in system-common memory. The typed data structures, which include binary and counting semaphores, queue semaphores, and ring buffers, are defined by the kernel and described in Section 2.2.1.

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

The CRST$ 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. See also the CRLN$ and CRSR$ primitives concerning logical-name and shared-region structure creation.

Syntax

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

Variant	Macro Call
CRST$	CRST$ [area,sdb,styp,satr,value]
CRST$S	CRST$S [sdb,styp,satr,value]
CRST$P	CRST$P [sdb,styp,satr,value]

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
```

styp

The type of structure to be created, as indicated by a predefined symbol. The valid structure-type symbols for a CRST$ call are:

ST.BSM—Binary semaphore
ST.CSM—Counting semaphore

ST.QSM—Queue semaphore
ST.RBF—Ring buffer
ST.UDF—Unformatted structure

The symbol values are defined by the QUEDF$ macro in the COMM and COMU system macro libraries. This argument has the form:

```
[STYP=]type-symbol
```

satr

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

SA$OFF—FIFO ordering **or**
SA$OPR—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)

SA$IFF—FIFO ordering **or**
SA$IPR—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 PELM$/PELC$ primitives (input access) and the GELM$/GELC$ primitives (output access). The access attributes are:

SA$RIR—Record-oriented input access
SA$RIS—Stream-oriented input access
SA$ROR—Record-oriented output access
SA$ROS—Stream-oriented output access

Note that SA$IFF and SA$IPR 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 also defined by the QUEDF$ macro. This argument has the form:

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

The argument value is not meaningful for an unformatted structure and must be #0 if type is ST.UDF.

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 unformatted structure, value is the even number of bytes to be allocated. This argument has the form:

```
[VALUE=]integer-value
```

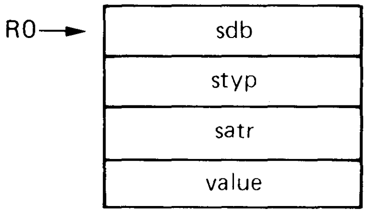
Restrictions

The first word of the passed SDB (the structure index) must be zeroed.

The minimum size of a ring buffer is 8 bytes; the maximum, 8128 bytes. The number of bytes specified (value argument) must be even.

Argument Block

The calling argument block generated (or assumed to exist) by the CRST$x macro has the following format:



MLO-426-87

Syntax Example

```
CRST$S sdb=#SEM,styp=#ST.BSM,satr=#SA$OPR,value=#1
```

Semantics

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

If the operation is unsuccessful, because of insufficient space in system-common memory, the primitive returns immediately to the caller, with the value FALSE in R0.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; an invalid structure type was specified.

ES$IPR Illegal primitive; user code attempted to create a PCB (type ST.PCB specified).

ES$SNI Structure name already in use; a kernel structure already exists with the specified name.

3.13 DAPK$ (Deallocate Packet)

Pascal equivalent: DEALLOCATE_PACKET Procedure

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

The inverse of DAPK$ is the ALPK$ primitive. ALPK$ lets a process 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]

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

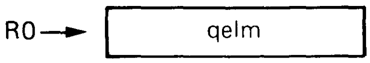
The address of the packet that is to be deallocated. This argument has the form:

```
[QELM=]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:



MLO-427-87

Syntax Example

```
DAPK$ area=#BLOCK,qelm=R3
```

Semantics

The DAPK$ primitive tests the pointer value to ensure that it is a valid packet pointer in the kernel's address space. If the packet pointer is valid, the primitive returns the packet to the kernel's free-packet pool. If no other process is waiting for packet allocation, DAPK$ 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 sequence may cause the calling process to be preempted, depending on the priority of the unblocked process. (See the ALPK$ primitive.)

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IAD Invalid address; the pointer value is not a word address within the range of valid packet addresses. (The address is checked only if the CHECK option is selected in the configuration file.)

3.14 DEXC$ (Dismiss Exception Condition)

Pascal equivalent: RELEASE_EXCEPTION Procedure

The Dismiss Exception Condition (DEXC$) primitive is used by an exception-handler process to return an exception-wait process to the kernel for additional disposition. The process changes to the appropriate ready state, normally ready active. (See Chapter 6 for a general discussion of exception handling.)

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 DEXC$ 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 service routine, if any.

Syntax

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

Variant	Macro Call
DEXC$	DEXC$ [area,pcb,action]
DEXC$S	DEXC$S [pcb,action]
DEXC$P	DEXC$P [pcb,action]

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 the PCB that is to be returned to the kernel. This argument has the form:

```
[PCB=]pcb-pointer
```

action

The type of action, as indicated by a predefined action code symbol, to be taken by the kernel. The action code symbols are:

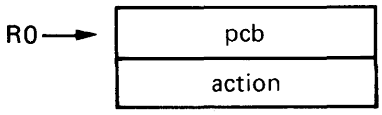
EA$DIS—Dismiss exception; place process in ready state.
EA$ABT—Abort process by forcing termination entry point.
EA$PAS—Pass exception to the process, if possible; otherwise, abort.

These symbols are defined by the EXACT$ macro in the COMM and COMU libraries. This argument has the form:

```
[ACTION=]action-symbol
```

Argument Block

The calling argument block generated (or assumed to exist) by the DEXC$x macro has the following format:



MLO-428-87

Syntax Example

```
DEXC$S pcb=R4,action=#EA$DIS
```

Semantics

The DEXC$ primitive places the passed PCB, which was received in exception-wait-active state on 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 the subject process was suspended while in the exception-wait state.) Return of the subject process to the ready-active state implies a scheduler call, which will cause the handler process to be preempted if the subject process has a higher priority.

The kernel will take one of the following actions relative to the subject process:

- Cancel the exception, allowing the process to be reentered normally when it is rescheduled (action = EA$DIS).
- Abort the process, causing its termination entry point to be forced when the process is rescheduled (action = EA$ABT).
- Pass the exception condition to the process's own exception service routine. If no exception entry point exists for the subject process 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 SERA$ primitive concerning exception handling within the process.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IPM Illegal parameter; an invalid PCB address or an invalid action code was specified.

3.15 DFSPC$ (Define Static Process)

Pascal equivalent: Declaration of a PROGRAM

The Define Static Process (DFSPC$) assembly-time macro defines the entry point and other attributes of a static process; the equivalent of a Pascal [SYSTEM(MICROPOWER)] PROGRAM declaration, which implicitly defines the source program's main body as a static process. Section 2.1.1 describes the characteristics of a MicroPower/Pascal static process. Each static process in an application is represented in the memory image by an entry in the static-process definition list. (Each list entry can be viewed as a prototype PCB.) At system start-up, the kernel's INIT routine uses the information in the entries to create static-process PCBs, which are queued on the ready-active state queue.

The DFSPC$ macro is not a primitive call. Rather, it generates a special read-only p-sect named .ALST. that contains the required static-process list entry. The complement of the DFSPC$ macro for run-time creation of dynamic processes is CRPC$. See the description of processes in Chapter 2 for additional information.

A static process in a mapped environment has a specific type of address mapping: general, device access, driver, or privileged. The mapping type is specified in the DFSPC$ macro in addition to the kind of information that is also specified in a CRPC$ call for a dynamic process. A dynamic process inherits its mapping type and its address mapping 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 2.1.7.

The PURE$, PDAT$, and IMPUR$ program-sectioning macros should be used with the DFSPC$ macro to segregate read-only code (ROM or RAM), read-only data (ROM or RAM), and read/write data (RAM only) program sections.

Syntax

The DFSPC$ macro syntax is:

```
DFSPC$ pid,pri,typ,cxo,grp,ter,cxl,sti,stl,sth,start,ini
```

pid

A 6-character ASCII string specifying the run-time name of the static process, that is, the name to be associated with the corresponding PCB. The name is padded with trailing blanks if it has fewer than six characters. This argument has the form:

```
[PID=]ascii-string
```

pri

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

```
[PRI=]integer-value
```

typ

The mapping type of the process, as indicated by a predefined process-type symbol. The type symbols are PT.GEN for general process mapping, PT.DEV for device-access process mapping, PT.DRV for driver process mapping, and PT.PRV for privileged (full-system) process mapping. The type symbols are defined by the PTDF$ macro in the COMM and COMU libraries. 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$FPP	FP-11 floating-point registers
CX$KT	MMU registers (optionally saved, always restored)
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. They are defined by the CXODF$ macro. This argument has the form:

```
[CXO=]option[!option]
```

grp

An integer code value of 1 to 255 indicating the exception-handling group to which the process belongs. The exception-group code value is significant only if one or more exception-handling processes are implemented in the application. Appropriate values are established by design convention. (See the CCND$ primitive and Chapter 6.) This argument has the form:

```
[GRP=]integer-value
```

ter

The entry point of the termination routine for the process. (See Semantics.) 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 value for the process's stack pointer (SP) register. Normally, this value will be the same as the sth argument value, assuming that the first-executed instruction affecting the stack is a push (autodecrement of SP). This argument has the form:

```
[STI=]first-top-of-stack-address
```

stl

The address of the low boundary of the user-allocated process stack, reserved for stack overflow checking. This argument has the form:

```
[STL=]low-bound-address
```

sth

The address of the high boundary of the user-allocated process stack, reserved for stack underflow checking. This argument has the form:

```
[STH=]high-bound-address
```

start

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

```
[START=]first-instruction-address
```

ini

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

```
[INI=]word-value
```

Restrictions

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. That is, the value of the user's SP register may range from sth (empty stack) to stl+2 (full stack). The kernel uses the stl and sth locations for dynamic stack-checking purposes, and those locations must not be modified by the user code. (See Semantics.)

The size of the process stack in bytes, excluding locations stl and sti, must equal or exceed the value $MINST, which defines the maximum number of bytes that the kernel and ISRs may push on the process stack. In unmapped systems, the value of $MINST is 54(decimal) bytes; in mapped systems, the value of $MINST is 0. When calculating the required stack space for a process, you should add the process's own stack requirement to $MINST.

Syntax Example

```
DFSPC$ pid=MOVER,pri=5,typ=PT.PRV,cxo=CX$KT,grp=1,ter=ABT,cxl=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 DFSPC$ macro generates the read-only program section .ALST., containing a static-process list entry. (The .ALST. p-sect must be the first section placed in the process image file built by the RELOC utility at build time. That ordering is normally achieved through RELOC's alphabetic sorting of program sections.) At build time, the MIB utility links the entries into a static-process definition list for the kernel. The list is used during system initialization by the kernel's INIT routine to "install" the defined static processes.

The implications of the DFSPC$ parameters that are not covered in Section 2.1 are described in the following paragraphs.

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

The CX$FPP option (cxo argument) allows a process using FP–11 floating-point instructions to have the contents of the floating-point processor registers saved and restored when it is context switched. (If the option is specified either in a target environment that does not support the FP–11 instruction set or for a process that does not use those instructions, the PCB will be larger than necessary in either case, and needless overhead will be incurred in the latter case.)

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

The CX$KT option causes the mapping registers to be saved during context switch-outs, allowing a process with privileged, driver, or device-access mapping to modify its mapping at process level. In an unmapped system, this option is meaningless. In a mapped environment, it incurs needless overhead if it is applied to a process that does not modify its mapping or does so only by means of the MAPW$ and UMAP$ primitives.

The kernel places guard words (special values) in the stl and sth locations and tests those guard words during context switch-outs. Modification of either the lower or the upper boundary location will cause an exception of type EX$RANGE, code ES$STO or ES$STU.

Error Returns

Not applicable; this macro is not executable.

3.16 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. Additional 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]

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
```

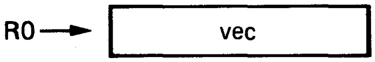
Restrictions

If connected, the specified vector must have been connected by the calling process.

A module that has a DINT$ primitive should not be added to a supervisor-mode library.

Argument Block

The calling argument block generated (or assumed to exist) by the DINT$x macro has the following format:



MLO-429-87

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 an illegal vector (ES$IVC) error.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IVC	Illegal vector; the specified vector address is invalid or points to a vector that is not connected or not owned by the calling process.

3.17 DLLN$ (Delete Logical Name)

Pascal equivalent: DELETE_LOGICAL_NAME Procedure

The Delete Logical Name (DLLN$) primitive allows the caller to eliminate the translation value defined for a given logical name, effectively "undefining" the name. More precisely, the DLLN$ primitive deletes the kernel data structure containing the translation string immediately associated with the name supplied in the call. (Contrast with the DLST$ primitive, which attempts to translate any logical name into the name of another type of structure and will not delete a logical-name structure. DLLN$, on the other hand, requires that the named structure be a logical-name value and will not perform any translation.)

The caller supplies the logical name and/or corresponding structure index in a structure descriptor block (SDB).

The complementary Create Logical Name (CRLN$) primitive defines the translation value associated with a logical name, and the Translate Logical Name (TRLN$) primitive returns the translation value associated with a currently defined logical name.

Syntax

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

Variant	Macro Call
DLLN$	DLLN$ [area,sdb]
DLLN$S	DLLN$S [sdb]
DLLN$P	DLLN$P [sdb]

area

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

```
[AREA=]arg-blk-address
```

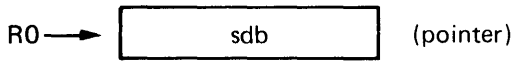
sdb

The address of a user-constructed SDB identifying the logical-name 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
```

Argument Block

The calling argument block generated (or assumed to exist) by the DLLN$x macro has the following format:



MLO-430-87

Syntax Example

```
DLLN$S sdb=#LGNAME
```

Semantics

The DLLN$ primitive verifies that the kernel data structure identified by the passed SDB is of type ST.LNM (logical name) and, if it is, deletes the structure and removes the corresponding name from the system name table.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such logical-name structure exists. (This error return could be caused by an erroneous SDB address.)

3.18 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.

The Delete Process service is transparent to the Pascal user; no predefined MicroPower/Pascal procedure is equivalent to the DLPC$ macro. In Pascal, deletion of a process is implicit when a PROCESS or a PROGRAM terminates; that is, 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

If the SM.ABI state-code modifier bit is not set in the caller's PCB (field PC.STS), the DLPC$ primitive removes the caller's PCB from the run queue, unlinks the PCB from the all-process list, deallocates the PCB (returns it to the kernel's free-memory pool) and calls the scheduler.

If the SM.ABI bit is set, indicating an abnormal-abort substate, the DLPC$ primitive switches the caller's PCB from the run queue to the inactive queue, sets the inactive state code (SC.IAC), unlinks the PCB from the all-process list, and calls the scheduler.

Error Returns

None.

3.19 DLRG$ (Deallocate Region)

Pascal equivalent: DEALLOCATE_REGION Procedure

The Deallocate Region (DLRG$) primitive allows the calling process to return a physical memory region, previously allocated by ALRG$, to the list of free-RAM segments maintained by the kernel. (See Section 5.3.) The base, size, and mode of the region to be deallocated aré specified by a region ID block (RIB) in the caller's address space. The primitive zeroes the size field contained in the RIB on successful deallocation. The mode of the region must be physical.

The DLRG$ primitive attempts to consolidate the free-RAM list whenever possible by combining the newly deallocated space with any adjoining space already represented in the list. Such consolidation results in a "new" free segment that is larger than the region just deallocated.

Whether list consolidation takes place or not, any region deallocation may free up enough space to allow a previously unsuccessful allocation request issued by another process to be satisfied if the request were reissued. DLRG$ always returns control to the calling process. (There is no blocking form of the complementary ALRG$ primitive.)

Although dynamic RAM allocation is designed primarily for a mapped target environment, the ALRG$ and DLRG$ primitives can be used in an unmapped application as well. RIB content differs between mapped and unmapped usage, as described for the ALRG$ primitive. (Presumably the RIB supplied to DLRG$ contains the values that were returned by an ALRG$ call.) Chapter 5 contains a general discussion of dynamic RAM allocation, including the use of DLRG$ in the context of the related primitives ACSR$, ALRG$, CRSR$, DLSR$, MAPW$, and UMAP$.

Syntax

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

Variant	Macro Call
DLRG$	DLRG$ [area,rib]
DLRG$S	DLRG$S [rib]
DLRG$P	DLRG$P [rib]

area

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

```
[AREA=]arg-blk-address
```

rib

The address of a 4-word (RI.SIZ bytes) area in user memory, the region ID block, that defines the region to be deallocated, as described under Semantics. This argument has the form:

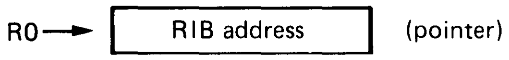
```
[RIB=]area-address
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the DLRG$x macro has the following format:



MLO-431-87

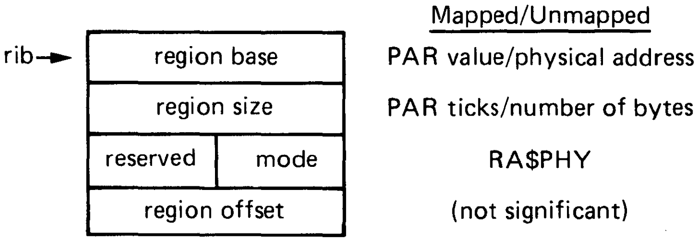
Syntax Example

```
DLRG$S rib=#8KBREG
```

Semantics

The DLRG$ primitive adds the memory space described by the caller's RIB to the kernel's linked list of free-RAM segments either by inserting a new list element or by modifying an existing one. (As a consequence, the information in the user's RIB is no longer valid.) If the RIB pointer is valid (even address) and the region-size field is nonzero, the DLRG$ primitive zeroes the size field, deallocates the described region, and returns control to the calling process. Otherwise, the primitive sets the caller's C bit and returns to the caller, with an error indication in R0.

The information in the caller's RIB area must be of the same form as that returned by a corresponding region-allocation operation, as follows:



MLO-432-87

The offset and size symbols defined for the RIB fields are:

RI.ADD	Region base
RI.LEN	Region size
RI.ATR	Region mode (attribute byte)
RI.RES	Reserved (high byte)
RI.OFF	Region offset
RI.SIZ	RIB size in bytes

The RIBDF$ macro in the MicroPower/Pascal COMM and COMU system macro libraries defines these symbols.

In a mapped environment, the region base must be a physical PAR value representing a 32-word physical boundary, not a virtual address. The region size specifies the number of consecutive 32-word units to be deallocated starting at the region base.

In an unmapped environment, the region base is simply the physical address of the region, and the region size is the number of bytes to be deallocated starting at the region base. If the specified size is not a multiple of 4, the next higher multiple of four bytes is deallocated.

In both cases, the region mode is indicated by the value of the symbol RA$PHY, denoting a physical region. (The RA$xxx mode symbols are defined by the RIBDF$ macro.) The last word of the RIB, the region-offset field, is not relevant for region deallocation and is ignored; the field is significant only in operations on shared common regions.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the RIB address is not on a word boundary.

ES$IPM Illegal parameter; the region-size value in the RIB is 0. (This error might reflect either an already deallocated region or an erroneous RIB address.)

Implementation Notes

The DLRG$ primitive does not limit the user to deallocating an entire region, as originally allocated, in a given operation. You can deallocate just a portion of a region or can return a region piecemeal in successive operations. Partial deallocation might be useful in some applications. Note, however, that it entails user modification of supplied RIB contents: the region base and size values supplied by ALRG$. In order to avoid obscure run-time problems, considerable care should be taken to ensure the correctness of any such modifications, since the primitive does minimal checking of RIB values. Any deallocation error introduced by user-modified values will corrupt the kernel's free-RAM list with unpredictable consequences; typically, a delayed system crash. The integrity of the free-RAM list depends entirely on the validity of the space descriptions supplied in deallocation requests.

3.20 DLSR$ (Delete Shared Region)

Pascal equivalent: DELETE_SHARED_REGION Procedure

The Delete Shared Region (DLSR$) primitive request lets the calling process delete the shared region descriptor (SRD) identified in the call; the kernel data structure that represents a region as shared. The effect of the operation is to preclude any subsequent access to the region through the ACSR$ primitive. However, the operation does not disable any previously gained access to the region.

Typically, the DLSR$ primitive would be used only in the termination routine of the process responsible for creating the SRD. (Processes commonly delete structures they have created if forced to terminate.) The kernel does not provide any automatic safeguard against inadvertent reference to a deleted (and possibly deallocated) shared region, since any process that previously accessed the region while it was shareable retains a description of it. The effective lifetime of a shared region could be coordinated among the processes having access to it through a special semaphore established for that purpose.

Chapter 5 contains a general discussion of region sharing, including the use of DLSR$ in the context of the related primitives ACSR$, ALRG$, CRSR$, DLRG$, MAPW$, and UMAP$. The CRSR$ primitive provides the complementary Create Shared Region operation, which declares a region as being shareable and assigns its run-time name.

Strictly speaking, there is no $DLSR primitive routine as such; the DLSR$ macro generates an appropriate call to the Delete Structure ($DLST) primitive routine, which implements the required operation.

Syntax

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

Variant	Macro Call
DLSR$	DLSR$ [area,sdb]
DLSR$S	DLSR$S [sdb]
DLSR$P	DLSR$P [sdb]

area

The address of a user-memory location at 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) that identifies the shared region to be deleted; that is, the SDB that contains the name and/or index and serial number of the corresponding kernel SRD structure. See Section 3.1.5 for the format and use of an SDB. This argument has the form:

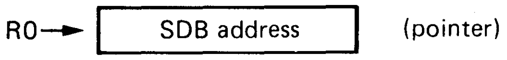
```
[SDB=]sdb-address
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the DLSR$x macro has the following format:



MLO–433–87

Syntax Example

```
DLSR$S sdb=#SRG8KB
```

This stack ($S) form of the macro call specifies the location of the structure descriptor block SRG8KB containing the name of the shared region to be deleted. See the CRSR$ primitive description for the corresponding region-creation example.

Semantics

The $DLST primitive routine, which is invoked by the DLSR$ primitive call, looks for a shared region descriptor (SRD) identified by the caller's SDB. If that SRD exists, the primitive deletes the SRD, removes the structure name from the system name table, and returns to the caller. If no such SRD exists, the primitive returns to the caller, with an error indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure description (index or name); no such shared region descriptor exists. (This error return could be caused by an erroneous SDB address.)

3.21 DLST$ (Delete Structure)

Pascal equivalent: DESTROY Procedure

The Delete Structure (DLST$) primitive deletes a specified semaphore, ring buffer, or unformatted structure from the system and deallocates the memory space associated with it. If a semaphore or ring buffer, 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 dynamic structure that is no longer needed. (See the DLLN$ and DLSR$ primitives concerning deletion of logical-name and shared-region structures.)

Syntax

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

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

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
```

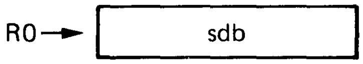
sdb

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
```

Argument Block

The calling argument block generated (or assumed to exist) by the DLST$x macro has the following format:



MLO-434-87

Syntax Example

```
DLST$S sdb=#SEM
```

Semantics

The DLST$ primitive checks the identified structure to ensure that it is not in use. (No in-use condition is defined for an unformatted structure.) 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 an in-use error indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IST Invalid structure descriptor (index or name); no such structure exists. (This error return could be caused by an invalid SDB address or by an SBD containing a logical name that does not translate into a valid nonlogical structure name.)

ES$SIU Structure is in use; the structure cannot be deleted in its current condition.

3.22 FORK$ (Fork Processing)

Pascal equivalent: None

The Fork Processing (FORK$) service request is used by an interrupt service routine (ISR) to discontinue execution at interrupt level and to resume as a fork routine at system level with all interrupts enabled. The execution of a fork routine is deferred until all pending interrupts have been serviced and any interrupted primitive operation has been completed but occurs before resumed-primitive execution and return to process level. The overall scheduling hierarchy is shown in Appendix A. The code following the FORK$ call becomes the body of the fork routine, which must terminate with a RETURN (RTS PC) instruction, as does a normal ISR that does not fork.

Fork routines provide a level of processing that is intermediate between interrupt level and process level. As discussed in Chapter 7, fork-level processing has two purposes: to permit an ISR to safely execute primitives and, when properly used, to minimize overall interrupt latency through deferral of less critical, interruptable I/O processing operations to that level.

An ISR must issue a FORK$ request before requesting any primitive service. Thus, the ISR will not violate kernel integrity by executing a primitive while a primitive operation has been interrupted, thereby causing the kernel to be reentered. (Kernel primitive operations can be interrupted but not reentered, and strictly sequential execution of primitives must be ensured.) The FORK mechanism guarantees sequential execution of primitives while permitting their use within ISRs, by serializing execution of ISR code segments that contain primitive requests. Note that the FORK service is not itself a primitive operation.

Fork routines have normal ISR context and a higher software priority than any process but, like primitives and processes, run at CPU priority 0. Like the ISR itself, the fork routine executes in kernel mode in a mapped environment. (A priority-7 ISR must issue a P7SYS$ service request, transforming itself into a normal ISR, before issuing a FORK$ request.) Fork routines are executed in FIFO order from a special queue that is independent of process scheduling.

An ISR that issues a FORK$ request can incur a fork-overrun error condition, resulting in an immediate return to the ISR at interrupt level with the C bit set. This error indicates that, at worst, a second interrupt has occurred before the fork routine for the first has begun execution or that, at best, a third interrupt has occurred before the fork routine for the second has begun execution. (See Semantics and Error Returns for more detail.) Depending on the kind of device being serviced and the inherent interrupt-handling capability of the target system, an overrun error may reflect an ISR design problem or may represent a temporary overload condition that is expectable and can be handled by the application through proper coding of the ISR and fork routine. (The fork routine can be made conditionally iterative based on indicators set by the ISR.)

Syntax

The syntax of the FORK$ macro call is:

```
FORK$
```

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 by an ISR executing at less than CPU priority-level 7 with normal ISR context. A priority-7 ISR must issue a P7SYS$ request before issuing a FORK$ request.

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

The fork routine should not execute any form of primitive that may block. (Violation of this rule is a likely cause of fork-overrun errors or a system crash.)

Semantics

The FORK service tests the fork block pointed to by R5 to ensure that it is not already linked into the fork queue. (The fork block is a portion of the IDB, so an ISR has only one fork block available to it for each vector that it services.) If the fork block is free, abbreviated ISR context (R3, R4, and PC) is saved in the block, and it is placed on the fork queue in FIFO order. 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 by the ISR when the FORK$ request is issued.

Before executing any resumed-primitive code and/or 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 an RTS PC instruction; the normal ISR exit procedure.

If the fork block pointed to by R5 is not free when the FORK$ request is issued, the FORK service sets the carry (C) bit and returns to the ISR.

Registers R0 through R5 are available for use after a successful $FORK request.

Error Returns

If the FORK$ call returns with the carry (C) bit set, the fork was unsuccessful because a previously queued fork routine is still pending. (At most, two forks can be outstanding, provided that the first fork routine has started execution, freeing the fork block for the following fork request.) The code immediately following the FORK$ call should test for the overrun error return and take appropriate action. If the C bit is set, the ISR is still running at interrupt level with its precall register values intact.

3.23 GELA$ (Get Element Any)

Pascal equivalent: GET_ELEMENT_ANY Procedure

The Get Element Any (GELA$) primitive implements a complex form of the Get Element operation; see the GELM$ and PELM$ primitives for a description of the basic Get Element and Put Element operations on ring buffers. GELA$ performs the basic Get Element operation on the logical OR of several ring buffers, with an optional timeout feature. That is, GELA$ permits the calling process to test for and, if necessary, wait on an available data record in any one of a set of ring buffers. Up to four ring buffers can be specified in the primitive request. (Each ring buffer must have record-mode output access.) If a complete record is immediately available in any of the specified ring buffers, the calling process gets the record and continues execution. Otherwise, the calling process blocks until one of the ring buffers can provide the requested number of bytes.

More specifically, if each of the specified ring buffers is initially empty or contains less than a full record, the caller blocks on all the buffers. (Partial data transfers never occur, because of the mandatory record-mode output access.) The process waits until the full request can be satisfied by any one of the ring buffers, at which point the process is unblocked from all of them.

Optionally, the Get Any operation can be terminated if a time interval specified in the request expires. On any nonerror return from the primitive (C bit clear), R0 will contain either an ordinal value identifying the ring buffer that satisfied the request or a 0, indicating that the request timed out.

Thus, the GELA$ primitive allows a process to get a specified number of elements from any of up to four ring buffers, although the primitive might be used primarily for its optional timeout capability, regardless of the number of ring buffers involved.

If a zero time period (immediate timeout) is specified in the request, the GELA$ primitive provides a complex form of the Conditional Get Element (GELC$) operation, which tests for available data but will not block the caller. See the GELC$ primitive for a description of the basic Conditional Get Element operation. Keep in mind, however, that only record-mode output applies in the case of a nonblocking GELA$ operation.

Syntax

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

Variant	Macro Call
GELA$	GELA$ [area,time,bufptr,bufcnt,sdb1,sdb2,sdb3,sdb4]
GELA$S	GELA$S [time,bufptr,bufcnt,sdb1,sdb2,sdb3,sdb4]
GELA$P	GELA$P [time,bufptr,bufcnt,sdb1,sdb2,sdb3,sdb4]

area

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

```
[AREA=]arg-blk-address
```

time

The address of a 2-word user-memory location that specifies a timeout interval, expressed in milliseconds. The first word of the double-precision integer contains the low-order portion of the time value; the second word (time+2) contains the high-order portion. An argument value of 0 implies no timeout for the request; the calling process may block indefinitely. This argument has the form:

```
[TIME=]word-address or #0
```

If the address value is nonzero but the time value pointed to is 0, the request will be timed out immediately if none of the specified ring buffers has an entire record when the primitive is called. That is, the calling process will never block if the specified time interval is 0.

bufptr

The address of the user's buffer that is to receive the data from the ring buffer. (The effective length of the buffer is implied by the bufcnt parameter value.) This argument has the form:

```
[BUFPTR=]buffer-address
```

bufcnt

The number of bytes to be transferred to the buffer pointed to by bufptr. (The value determines the record length for the operation.) This argument has the form:

```
[BUFCNT=]integer
```

sdb-i

The address of a structure descriptor block (SDB) that identifies one of the ring buffers to be operated on. From one to four SDB addresses may be specified. The order in which the SDBs are specified (or are identified if enumerated by keyword) determines the order in which the corresponding ring buffers are initially tested for data elements. (That order can be critical under certain real-time conditions, as discussed under Implementation Notes below.) The sdb-i arguments have the form:

```
[SDBi=]sdb-address
```

The value "i" may be 1 through 4 if the keyword form of argument is used.

Restrictions

Each ring buffer's output-access mode must be record. (See the satr argument of the CRST$ primitive.)

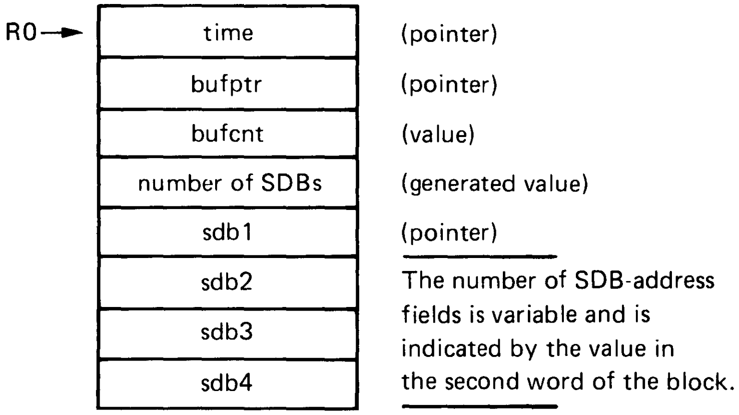
The number of bytes requested (bufcnt parameter) must not exceed the size of any ring buffer specified in the request.

The time-out value may not exceed (2**31)–1, the largest positive integer expressible in 32 bits. That is, the sign bit of the time-interval doubleword (bit 15 of the high-order word) must not be set. (The maximum valid value, in milliseconds, permits a time-out period of just over 24.89 days; see the SLEP$ primitive for more detail.)

In the keyword form of macro call, higher-numbered sdb-i keywords may not be used unless each of the lower-numbered sdb-i keywords is specified. That is, if the keyword sequence contains SDB3=, for example, the sequence must also include SDB1= and SDB2=, though not necessarily in numeric order.

Argument Block

The calling argument block generated (or assumed to exist) by the GELA$x macro has the following format:



R0→	time	(pointer)
	bufptr	(pointer)
	bufcnt	(value)
	number of SDBs	(generated value)
	sdb1	(pointer)
	sdb2	The number of SDB-address fields is variable and is indicated by the value in the second word of the block.
	sdb3	
	sdb4	

MLO–435–87

Syntax Example

```
GELA$ #GEARGS,#RESET,#INPBUF,#RECLEN,#PORTC3,#PORTCO,#PORTC1,#PORTC2
```

Semantics

For clarity, the following description ignores the unlikely case of multiple waiters for ring buffer output. That is, the description assumes that only one process is attempting to get data from a given ring buffer, although the GELA$ operation allows for the possibility of multiple getters and guarantees sequential access, as does the basic GELM$ primitive. The GELA$ primitive tests each ring buffer specified in the request for two conditions: at least bufcnt elements available or fewer than bufcnt elements available. (The ring buffers are tested in the order in which they are identified in the call, by either position or keyword value.) If any of the ring buffers contains at least bufcnt bytes at the time of the call, the primitive transfers bufcnt bytes from the first such ring buffer encountered and returns to the caller, with a nonzero value in R0. The R0 value, an integer between 1 and 4, indicates that the nth ring buffer identified in the call satisfied the request.

If all the ring buffers contain fewer than bufcnt elements and a zero timeout value was supplied in the call, the primitive returns immediately to the caller, with a zero value in R0, indicating a return that is due to timeout. (The calling process thus never leaves the run state in the case of an immediate-timeout form of request.)

If all the ring buffers contain fewer than bufcnt elements and either a zero time argument or a nonzero timeout value was supplied in the call, the primitive switches the calling process to the wait-active state. The process is blocked on each and every ring buffer specified in the request. The calling process remains blocked on all the buffers until at least bufcnt bytes of data (a full record) accumulates in any one of them. At that point, the primitive performs the requested data transfer, unblocks the caller from all the ring buffers, and switches the caller to the ready-active state, with the nonzero ordinal value in R0, as described above.

In the case of process blocking described above, if a nonzero timeout value was supplied in the call, the calling process is also blocked on an internal timer queue, as well as on one or more ring buffers. If the specified timeout period expires at any point before the request can be satisfied, the caller is removed from all blocking structures and is switched to ready-active state, with 0 in R0. Any partial record(s) accumulated at that point remain in the respective ring buffer(s).

In all cases described above, the user's C bit is cleared, distinguishing the value returned in R0 from an error-return indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; timer-value pointer is an odd address, or a buffer or SDB address is not on a word boundary or not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; either the bufcnt value exceeds the size of one of the ring buffers specified in the request or the timer value is out of range.

ES$IPR Invalid primitive; the output-access mode of one of the ring buffers specified in the request is stream, not record.

ES$IST Invalid structure description (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

Implementation Notes

Since the initial test of the ring buffers for a complete record is performed in determinate order, the order in which multiple buffers are identified in the call can be critical under certain real-time conditions. For example, if the relative frequency of Puts is high for one of several ring buffers and the "fast" ring buffer is identified as being first, either by position in the SDB sequence or by the keyword SDB1=, that ring buffer will tend to mask off the others in a sequence of GELA$ operations. In this case, the "slower" ring buffers may seldom or never be tested and serviced. Optimally, then, the ring buffer with the highest expected Put rate should be identified as last, the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the ring buffers are identified could be rotated in successive calls so that at least n buffers are guaranteed to be tested in n calls to GELA$.

As a contrary example, assume that the specified set of ring buffers represents device inputs (the common use) and that one of the devices has the highest priority in terms of its need to be serviced. (The service priority might be independent of expected input rates, which, if different, could be reflected by differing ring buffer sizes, for example.) In this case, the highest-priority buffer would be identified as first in the GELA$ call, making sure that the buffer is always tested on any call.

The correct or best-case strategy depends on application-specific factors, of course.

3.24 GELC$ (Conditional Get Element)

Pascal equivalent: COND_GET_ELEMENT Function

The Conditional Get Element (GELC$) primitive implements a nonblocking form of Get Element operation; compare with the unconditional GELM$ primitive. GELC$ 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. Informally, the meaning of a GELC$ request for a record-mode buffer is "get me n bytes right away or none at all," and the meaning for a stream-mode buffer is "get me as many bytes as you can, up to n."

In either case, however, GELC$ returns to the caller, with a value in R0 indicating how many bytes are still needed to fully satisfy the request. Thus, a return value of 0 indicates that the request has been fully satisfied; the number of bytes specified in the call have been 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 CRST$ primitive. For a ring buffer with record-mode output, GELC$ 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, GELC$ attempts to satisfy the request with either a full or a partial transfer. That is, the primitive gets as many bytes as are immediately 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 GELM$ request to be satisfied, no bytes are available for the caller.

The complementary PELM$ and PELC$ primitives allow a process to put bytes into a ring buffer.

Syntax

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

Variant	Macro Call
GELC$	GELC$ [area,sdb,bufptr,bufcnt]
GELC$S	GELC$S [sdb,bufptr,bufcnt]
GELC$P	GELC$P [sdb,bufptr,bufcnt]

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. 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:

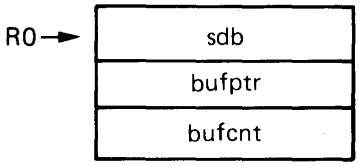
```
[BUFCNT=]integer
```

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 always fail, since the buffer will never contain a full record.)

Argument Block

The calling argument block generated (or assumed to exist) by the GELC$x macros has the following format:



MLO–436–87

Syntax Example

```
GELC$S sdb=#TTRING,bufptr=#LOW,bufcnt=#10.
```

Semantics

If the specified ring buffer's output-access attribute is SA$ROR (record mode), the GELC$ 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 fewer than bufcnt bytes, GELC$ returns immediately, with the value bufcnt in R0, indicating that no bytes were transferred.

If the specified ring buffer's output-access attribute is SA$ROS (stream mode), GELC$ 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.

A successful GELC$ operation may cause preemption of the caller if the operation unblocks a process waiting to Put elements (see Section 2.1.4). That is, return from a successful GELC$ request is not necessarily immediate.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

3.25 GELM$ (Get Element)

Pascal equivalent: GET_ELEMENT Procedure

The Get Element (GELM$) primitive extracts a specified number of data bytes from a ring buffer and transfers them to the caller's buffer area. If the ring buffer does not contain enough bytes to satisfy the Get request, the calling process blocks on the ring buffer, waiting for a sufficient number of bytes to become available.

In general, if two or more processes are getting data from the same ring buffer, the calling process will block if another process is waiting for its Get request to be satisfied. The calling process must wait its proper turn (whether by FIFO or priority order) for access to the buffer, since sequential access to a ring buffer is ensured for multiple readers as well as for multiple writers. The process that blocks first is given active read access to the buffer and is never displaced by another, higher-priority process, regardless of the ordering attribute of the waiting-output-process list.

If the ring buffer's output-access mode is stream, the data transfer may occur in increments while the process is waiting. Stream-mode access permits a ring buffer to be smaller than the "records" that may be passed through it.

The complementary PELM$ and PELC$ primitives allow a process to put elements into a ring buffer.

The conditional, nonblocking form of GELM$ is the GELC$ primitive.

Syntax

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

Variant	Macro Call
GELM$	GELM$ [area,sdb,bufptr,bufcnt]
GELM$S	GELM$S [sdb,bufptr,bufcnt]
GELM$P	GELM$P [sdb,bufptr,bufcnt]

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. 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:

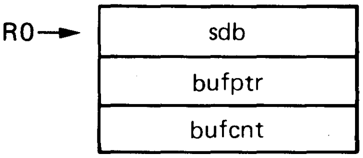
```
[BUFCNT=]integer
```

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.

Argument Block

The calling argument block generated (or assumed to exist) by the GELM$x macros has the following format:



MLO-437-87

Syntax Example

```
GELM$ area=#ARGBLK,bufptr=#BUF1,bufcnt=#SIZE
```

Semantics

If no other process is waiting to get data from the specified ring buffer, the GELM$ primitive tests the buffer for bufcnt bytes of available data. If at least that amount of data is available, the primitive transfers the requested number of bytes from the ring buffer to the caller's buffer and returns control to the caller. (A Get operation effectively deletes the corresponding data from the ring buffer.)

If the ring buffer contains fewer than bufcnt bytes and its output access mode is record (SA$ROR), the primitive blocks the caller with active read access to the ring buffer and calls the scheduler. When a full record becomes available as a result of one or more subsequent Put Element operations, the transfer is performed, and the waiting process is unblocked. If the ring buffer contains fewer than bufcnt bytes and its output access mode is stream (SA$ROS), the primitive blocks the caller with active read access to the ring buffer, 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 (possibly by a series of partial transfers) and the waiting process is unblocked.

Note

> In stream mode, partial transfers from a ring buffer (incremental Get operations) can occur in units of at least one-quarter of the buffer size, up to the point where the full Get request can be satisfied. Conversely, partial transfers to a ring buffer (incremental Puts) can occur whenever three-quarters or more of the buffer is empty, up to the point where the full Put request can be satisfied.

If one or more processes are already waiting to get data from the ring buffer at the time of the call, implying that another process has active read access, the calling process is blocked on the buffer's waiting-output-process list in either FIFO or priority order, depending on the ring buffer definition. (A process with active access is never displaced by another process, regardless of relative priorities.) The process waits its turn to gain active read access, at which point it is treated as described above.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

ES$IPM Illegal parameter; the bufcnt value exceeds the size of the ring buffer for a record-mode operation.

3.26 GMAP$ (Get Mapping)

Pascal equivalent: GET_MAPPING Procedure

The Get Mapping (GMAP$) primitive allows the calling process to obtain a copy of its own current mapping or that of any other specified process. GMAP$ returns the mapping information stored in the mapping-context restore area associated with the PCB of the subject process to a buffer area specified in the call.

The mapping information consists of 16 words of Page Address Register (PAR) and Page Descriptor Register (PDR) values unless I&D-space separation is in effect for the subject process, in which case 32 words of information (values for both the instruction and data Active Page Register (APR) sets) are returned. (Separate I&D-space mapping is possible on an LSI–11/73 or similar target system but is not necessarily in effect for a given process.)

Syntax

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

Variant	Macro Call
GMAP$	GMAP$ [area,pdb,buf]
GMAP$S	GMAP$S [pdb,buf]
GMAP$P	GMAP$P [pdb,buf]

area

The address of a user-memory location at 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 subject process, or 0. If 0 is specified as the argument value, the calling process is implied. (See Section 3.1.6 for the format and use of a PDB.) This argument has the form:

```
[SDB=]pdb-address or #0
```

buf

The address of a 16- or 32-word user-memory area in which the mapping information is to be returned by the primitive. This argument has the form:

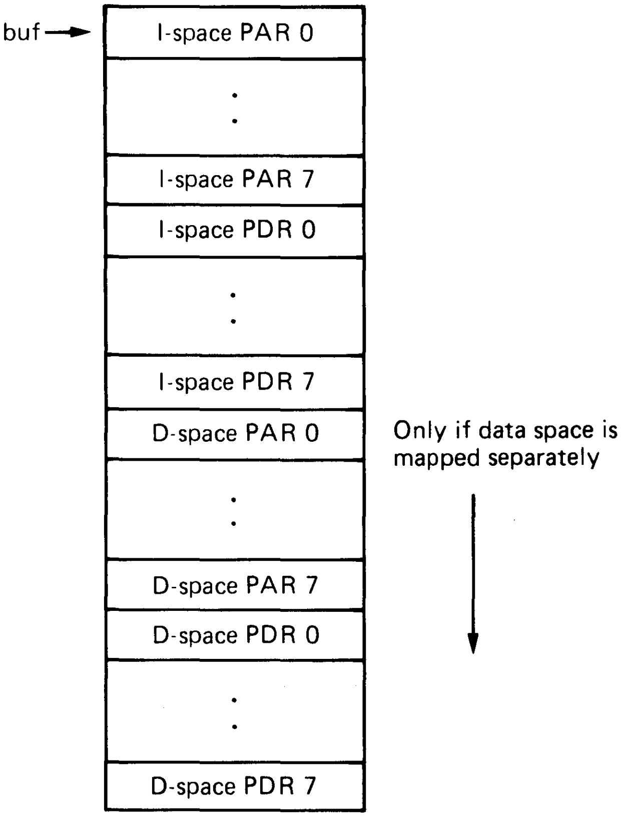
```
[BUF=]buffer-address
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Format of Information Returned

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

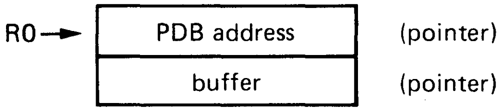


MLO–438–87

The "I-space" mapping registers refer to the only set of user APRs in a target system that does not provide differentiated instruction and data mapping, such as an LSI–11/23. Note that the PDR word of any unmapped (currently unused) APR will contain 0; the content of the corresponding PAR is undefined and is not significant.

Argument Block

The calling argument block generated (or assumed to exist) by the GMAP$x macro has the following format:



MLO–439–87

Syntax Example

```
GMAP$S pdb=#0,buf=#MAPBUF
```

This stack ($S) form of the macro call requests that the caller's mapping context be returned to the user-memory area beginning at location MAPBUF.

Semantics

The GMAP$ primitive copies the contents of the mapping-context restore area pointed to by the subject process's PCB (field PC.MAP) to the buffer area specified in the call.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or PDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure description (index or name); no such process control block exists. (This error return could be caused by an erroneous PDB address if address checking is not in effect.)

ES$IPR Illegal primitive call; the primitive service was requested in an unmapped environment.

Applications

Among other possible uses, the GMAP$ primitive allows a process with general mapping to inspect its current mapping in order to identify unused APRs for use in dynamic mapping operations. This in turn allows the process to optimize a sequence of mapping/remapping operations by using the fixed mode of MAPW$ call, which eliminates the need for intervening UMAP$ calls.

3.27 GTIM$ (Get Time)

Pascal equivalent: GET_TIME Procedure

The Get Time (GTIM$) primitive returns the approximate system time in milliseconds, assuming that a system clock is present and configured on the target system. System time is either of the following:

- The elapsed time since the last system initialization (zero based).
- The base time set by the STIM$ primitive plus the elapsed time since the base system time was last set. (A base time, if used, is normally set as part of the system start-up or restart procedures.)

GTIM$ returns the system time as a triple-precision integer in a 3-word area specified by the caller. The triple-precision, 48-bit value allows for a very large maximum elapsed time (until December 31, 2099), if the Pascal-implemented date/time setting algorithm is used. The calling process may need to consider only the low-order or low- and middle-order portions of the time value.

The kernel calculates system time in milliseconds on the basis of interrupts from a clock source of 50, 60, 100, or 800 Hz. Thus, the time is kept in multimillisecond "granules," or clock ticks. (A 60–Hz clock, for example, "ticks" only once every 16.6666667 milliseconds.) Thus, a range of possible discrepancy between reported system time and actual elapsed time varies with clock frequency. Assuming that the calling process is not preempted before it has a chance to use the value reported by $GTIM, the worst-case discrepancy between the actual elapsed time and the reported system time due solely to clock-tick granularity is:

- 50 Hz—20 milliseconds
- 60 Hz—17 milliseconds
- 100 Hz—10 milliseconds
- 800 Hz—2 milliseconds

Also, if the caller is preempted during or just following the Get Time operation, the duration of the preemption will add to the margin of error in the reported time as perceived by the calling process. As a safeguard against the preemption problem, the calling process might temporarily raise its priority across the $GTIM call and the code that processes the returned value.

The SLEP$ primitive provides a related process blocking-and-wakeup service based on elapsed system time.

The STIM$, GTIM$, and SLEP$ primitives together replace the functionality previously provided by the DIGITAL-supplied Clock Service Process, which is now obsolete.

Syntax

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

Variant	Macro Call
GTIM$	GTIM$ [area,tim]
GTIM$S	GTIM$S [tim]
GTIM$P	GTIM$P [tim]

area

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

```
[AREA=]arg-blk-address
```

tim

The address of a 3-word area in user memory in which the primitive returns the system time value, as described under Semantics. This argument has the form:

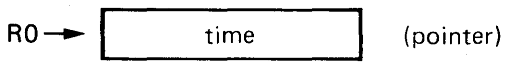
```
[tim=]area-address
```

Restrictions

The time argument must specify an even address.

Argument Block

The calling argument block generated (or assumed to exist) by the GTIM$x macro has the following format:



MLO–440–87

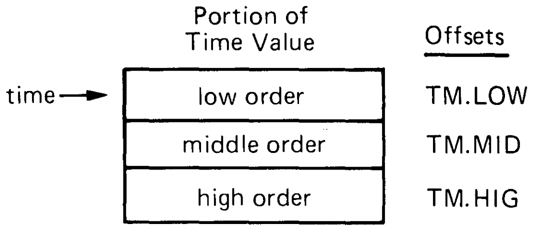
Syntax Example

```
GTIM$S tim=#TIMVAL
```

Semantics

The GTIM$ primitive disables interrupts, moves the 3-word system time value to the area specified in the call, enables interrupts, and returns to the caller.

GTIM$ returns the system time to the caller's time area in the following form:



MLO-441-87

The TM.xxx offset symbols used by the kernel are defined by the TIMDF$ macro.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IAD Invalid address; the time address is an odd value.

3.28 GTST$ (Get Process State)

Pascal equivalent: GET_STATE Procedure

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

Since process state information is dynamic, it could be invalid by the time it is available to the caller, because of possible effects of interrupt servicing. Unless the calling process is checking its own substate, however, this possibility is likely to be of concern only if the information is about another process.

Syntax

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

Variant	Macro Call
GTST$	GTST$ [area,pdb,buf]
GTST$S	GTST$S [pdb,buf]
GTST$P	GTST$P [pdb,buf]

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 reported on, 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 or #0
```

If a PDB is specified and its structure index and name fields are zeroed, the calling process is also implied, and the structure ID of the caller's PCB is returned in the PDB.

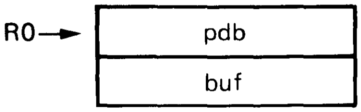
buf

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:

```
[BUF=]buffer-address
```

Argument Block

The calling argument block generated or assumed to exist by the GTST$x macros has the following format:



MLO–442–87

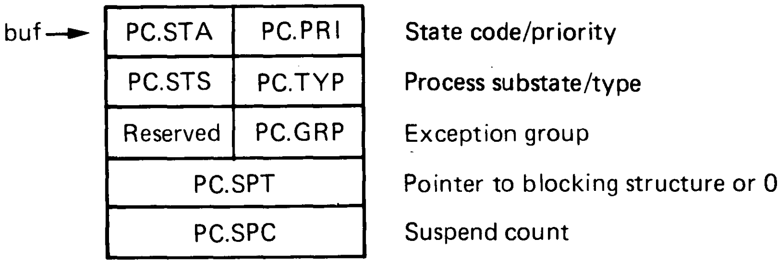
Syntax Example

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

Semantics

The GTST$ primitive copies five words of status information from the PCB of the subject 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:



MLO–443–87

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or PDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); the passed PDB does not correctly describe or imply a process. (This error return could be caused by an invalid PDB address if address checking is not in effect.)

3.29 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.

Since the value of a structure may change immediately after it is inspected, the information this primitive provides must be used with caution in order to prevent the introduction of race conditions. (GVAL$ will also return the type code, only, for a PCB or unformatted structure.)

An alternative use of GVAL$ provides information about the target hardware configuration. In this variant usage, provided primarily for system processes, GVAL$ returns two words of configuration information kept in kernel pure-data space.

Syntax

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

Variant	Macro Call
GVAL$	GVAL$ [area,sdb,type,val]
GVAL$S	GVAL$S [sdb,type,val]
GVAL$P	GVAL$P [sdb,type,val]

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, or 0. If #0 is specified, indicating "no structure," hardware configuration information is implied. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

```
[SDB=]sdb-address or #0
```

type

The address of the location in which the structure type code is to be returned by the primitive. (Alternatively, a bit mask reflecting hardware options is returned in the specified location.) 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. (Alternatively, a value reflecting the system clock frequency is returned in the specified location.) This argument has the form:

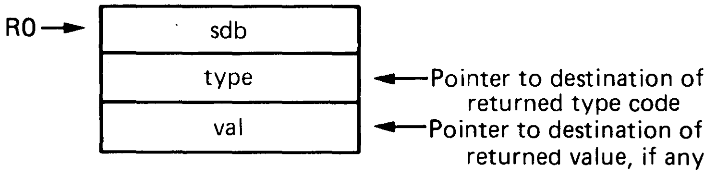
```
[VAL=]word-address
```

Restrictions

The passed SDB must not contain a logical name (or otherwise identify a logical-name structure (ST.LNM)) that does not translate into the name of a structure of a type other than ST.LNM or ST.SRD.

Argument Block

The calling argument block generated (or assumed to exist) by the GVAL$x macro has the following format:


R0
sdb
type
val
Pointer to destination of returned type code
Pointer to destination of returned value, if any

MLO-502-87

Syntax Example

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

Semantics

If the SDB argument value is not 0, the GVAL$ primitive inspects the type and value of the specified structure, stores the type code and structure value in the user locations (type and val) pointed to by the call, and returns control to the caller.

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

ST.BSM	Binary semaphore
ST.CSM	Counting semaphore
ST.QSM	Queue semaphore
ST.RBF	Ring buffer
ST.SRD	Shared region descriptor
ST.PCB	Process control block
ST.UDF	Unformatted structure

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

- For ST.BSM, the value of the gate variable (0 or 1)
- For ST.CSM, the count of pending signals (0 or positive)
- For ST.QSM, the count of queued packets (0 or positive)
- For ST.RBF, the count of data bytes in the ring buffer
- For ST.SRD, ST.PCB, or ST.UDF, no significance (0)

If the SDB argument value is 0, the GVAL$ primitive returns two kernel values, reflecting build-time configuration options, in locations type and val, as follows:

- In the caller's type location, a bit-mask word indicating the possible target processor and bus characteristics (kernel pure-data word $CNFIG)
- In the caller's val location, an integer value specifying the frequency of the system clock (50, 60, 100, or 800 Hz) or 0 for no clock (kernel pure-data word $CLKFQ)

The format of the configuration mask value returned at location type is as follows:

Bit Position	**Bit-Mask Symbol**	**Significance If Bit Is Set**
0	HC$FPP	FP–11 floating-point processor
1	HC$FIS	FIS instruction set
2	HC$F11	F–11 microprocessor, as in an LSI–11/23
3	HC$J11	J–11 microprocessor, as in an LSI–11/73, PDP–11/83, MicroPDP–11/53, or KXJ11–CA
4	HC$T11	T–11 microprocessor, as in a FALCON or FALCON–PLUS
5	HC$IOP	KXT11–CA or KXJ11–CA peripheral processor
6	HC$Q22	22-bit bus addressing
7	HC$MMU	Memory mapping enabled
8	HC$CMR	CMR21 target system
9	HC$FPA	Floating-point accelerator processor
.	.	
.	.	Reserved
.	.	
15	HC$ROM	ROM/RAM memory image

The HC$xxx symbols, defined by the HDCDF$ macro, represent single bit values, which may be used to test corresponding bit positions in the configuration word. Note that an LSI–11/2 target is implied if bits 2 through 5 are clear.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IPR Illegal primitive; the SDB contained a logical name that did not translate to a valid structure name.

ES$IST Invalid structure descriptor (index or name); no such semaphore, ring buffer, PCB, or unformatted structure exists. (This error return could be caused by an invalid SDB address.)

3.30 IMPUR$ (Define an Impure-Data Program Section)

Pascal equivalent: None

The IMPUR$ macro declares a program section of impure data within a MicroPower/Pascal source module. A program section declared with IMPUR$ has the read/write, data, relocatable, and concatenated attributes; has the name .MDAT.; and must be allocated in RAM. (The MIB utility will disallow any inadvertent attempt to place it in ROM.)

The IMPUR$ macro is not a primitive call. Rather, it is an assembly-time macro that is used with the DFSPC$, PURE$, and PDAT$ macros to segregate read-only code (ROM or RAM), read-only data (ROM or RAM), and read/write data (RAM only) program sections. During system building, program sections that have been declared with the DFSPC$, PURE$, PDAT$, and IMPUR$ macros (or by other means) are grouped according to their read-only versus read/write attribute and consolidated by p-sect name by the RELOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.) If instruction- and data-space separation is requested for the static process at build time, RELOC also groups p-sects according to their instruction versus data attribute, so use of both PURE$ and PDAT$ (or their equivalents) is required for that case, as well as IMPUR$. Although the use of PURE$, PDAT$, and IMPUR$ is not mandatory (if equivalent program sectioning is achieved by other means), these macros are convenient, and their use is recommended.

Note that program sectioning is implicit in Pascal source programs. That is, appropriate 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 .MDAT. p-sect definition with the attributes RW (read/write), D (data), REL (relocatable), CON (concatenated), and LCL (local). The GBL/LCL attribute distinction is not significant for MicroPower/Pascal applications.

Error Returns

Not applicable; this macro is not executable.

3.31 MAPW$ (Map Window)

Pascal equivalent: MAP_WINDOW Procedure

The Map Window (MAPW$) primitive, valid only in a mapped environment, permits a process to associate a sequence of virtual addresses with a specified region of physical memory. More precisely, MAPW$ allows the calling process to extend or modify its virtual-to-physical mapping to include a previously unmapped area of physical memory. The caller supplies the physical description of a memory region, through a region ID block (RIB), and specifies the portion of the region to be mapped. The MAPW$ primitive uses this information to alter the calling process's MMU registers and PCB mapping context, normally by modifying one or more currently unused APRs, and returns an appropriate virtual-address value to the caller. (Optionally, you can choose the APR or sequence of APRs to be modified.) Thus, the process obtains a virtual-address window into a region of memory that was not in its original address space. The region may be a private physical region allocated to the process by the ALRG$ primitive or may be a shared common or physical region previously accessed through the ACSR$ primitive.

The UMAP$ primitive provides a complementary unmapping operation, which may be required between successive mapping operations, depending on the mode of MAPW$ usage. The principal application objectives for the MAPW$ and UMAP$ primitives are:

- Usability by a process with general mapping, which cannot otherwise alter its mapping context. (Other types of processes, which can perform direct MMU modification, may use MAPW$ to alter mapping without the need for MMU-register saving during context switchouts, a performance consideration. The reason for the use of MAPW$ is that, in contrast to the use of direct MMU modification, changes are also made in the mapping-context restore area associated with the caller's PCB.)
- Use in conjunction with the ALRG$ or ACSR$ primitives, which provide the physical description of a memory region in the required format.

The MAPW$ primitive is described here in terms of that primary application context. MAPW$ and UMAP$ can also be used by processes with device-access, privileged, or driver mapping, of course, and also for mapping of objects other than regions as such.

Assuming that a process with general mapping does not "borrow" (force remapping of) an already allocated APR, the minimum requirement for using MAPW$ is that the calling process's statically allocated virtual-address space does not exceed 28K words. In other words, at least one of the static process's APRs must remain unused, or inactive, at build time. This requirement can be overridden by the "fixed" APR option, which forces MAPW$ to use an APR indicated by the caller rather than the first unused APR found by the primitive. (Note that a dynamic process inherits the entire address space of its parent process and might not need the full extent of its inherited mapping.)

The size of a window is controlled by a user-specified length parameter, which implies the number of APRs needed for the window. Thus, a process can map to an entire multipage region in a single operation, given that enough APRs are available for modification. If the caller does not have multiple APRs available for the window and the region to be mapped is larger than 4K words (one virtual page), the process can step through the region by repeated mappings of a single APR, using suitably incremented window offsets. The potential size of a window is constrained only by the number of contiguous APRs available for the mapping, not by the size of the region as described in the RIB. Therefore, you should ensure that the requested window length not cause the window to extend beyond the end of the region, as a protection against inadvertent access to space beyond the region that is due to a coding error.

The information supplied in the RIB that is pointed to in the call specifies the region's location (physical base and byte offset, if any), size, and common/physical mode attribute. (The content of the RIB is assumed to be that returned by a prior ALRG$ or ACSR$ call; the format of the information is as described for those primitives.) In addition to the RIB, the caller supplies the length to map and an optional additive offset into the region specified in PAR ticks (32-word units); typically, a multiple of 128 ticks when stepping through a large region with a single-PAR window.

The combination of those parameters determines the size and positioning of the mapped window within the region for a given call. The RIB content is never modified by the MAPW$ primitive; the physical description of the region remains invariant throughout successive, incremental remappings. In general, user modification of the RIB content user is also discouraged as an unsafe practice.

Chapter 5 contains a general discussion of region sharing and mapping, including the use of MAPW$ in the context of the related primitives ACSR$, ALRG$, CRSR$, DLRG$, DLSR$, MAPW$, RCTX$, SCTX$, and UMAP$. The ACSR$ and ALRG$ primitives provide the supporting operations that obtain RIB information. The GMAP$, SCTX$, and RCTX$ primitives provide additional support for mapping operations that involve "borrowing" of one or more APRs.

Syntax

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

Variant	Macro Call
MAPW$	MAPW$ [area,rib,len,opt,offset,wptr]
MAPW$S	MAPW$S [rib,len,opt,offset,wptr]
MAPW$P	MAPW$P [rib,len,opt,offset,wptr]

area

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

```
[AREA=]arg-blk-address
```

rib

The address of the region ID block in user memory containing the physical description of the region to be mapped to. This argument has the form:

```
[RIB=]area-address
```

len

An unsigned integer representing the desired size of the virtual window (the length to map to) in bytes. This argument has the form:

```
[LEN=]integer
```

opt

A set of paired, predefined, bit-mask symbols specifying optional features of the mapping operation. (The logical OR of the symbol values and/or defaults produces a bit-mask word in the calling argument block.) The option symbols, defined by the RIBDF$ macro, and their meanings are:

WD$INS—The operation modifies the process's I-space mapping **or**
WD$DAT—The operation modifies the process's D-space mapping, the default.

Note that the WD$INS/WD$DAT alternatives are meaningful only if I&D-space separation is in effect for the process (possible in an LSI-11/73 target).

WD$RO—The operation maps the window for read-only access **or**
WD$RW—The operation maps the window for read/write access, the default.

WD$FIX—The.operation modifies the APR(s) determined by the precall, virtual-address value in location wptr **or**
WD$FRE—The operation modifies the free APR(s) chosen by the primitive, the default. (See Use of the WPTR Parameter for more details.)

WD$LEC—The operation leaves caching as is, either enabled or disabled (default) **or**
WD$DAC—The operation disables caching for this window. Sets bit 15 of each PDR to disable caching for each APR.

This operation is necessary on the arbiter side when you map to a KXJ shared memory area, if the arbiter uses cache memory; but even if this symbol is specified on an arbiter processor not using cache memory, there are no adverse effects. See Appendix B of the *MicroPower/Pascal I/O Services Manual* for additional information.

Two or more option symbols may be ORed as required. This argument has the form:

```
[OPT=](option-symbol[!option-symbol ...])
```

The argument can be null, implying the option defaults WD$DAT, WD$RW, WD$FRE, and WD$LEC.

offset

An integer representing the desired displacement of the virtual window from the beginning of the region, expressed in PAR ticks (32-word units). This argument is used when "stepping through" a large region by incremental remapping of a window. This argument has the form:

```
[OFFSET=]integer
```

wptr

The address of a word in user memory (the "window pointer") in which the primitive returns a virtual address corresponding to the first location in the mapped window, fully adjusted for offset(s) as described under Semantics. If the WD$FIX option is not specified, the precall value of location wptr is not significant, and the primitive chooses the APR(s) to be used in the mapping operation. If WD$FIX is specified, however, the precall value of wptr is used by the primitive to select the first or only APR to be modified, as described under Use of the WPTR Parameter. This argument has the form:

```
[WPTR=]word-address
```

Restrictions

This primitive may be used only at process level; it may not be called from an ISR fork routine.

If I&D-space separation is in effect for the calling process, the combination of the WD$INS option and the WD$RW (default) option is invalid.

You cannot use the MAPW$ primitive to modify APR 0 of a process without I&D-space separation or D-space APR 0 of a process with I&D-space separation if that process accesses a supervisor-mode library.

If the free (default) mode of APR selection is used, Unmap Window calls are required between successive Map Window calls for iterative remapping of a window.

Use of the WPTR Parameter

In general, if the default (free mode of window mapping) is used, wptr is a destination-only location, but if fixed mode is selected, location wptr is both a source and a destination.

More specifically, if WD$FRE is specified or defaulted in the call, the precall content of location wptr is ignored by the primitive, and the primitive selects one or more free APRs for the mapping operation. The virtual address returned at wptr reflects the first or only APR selected for the window.

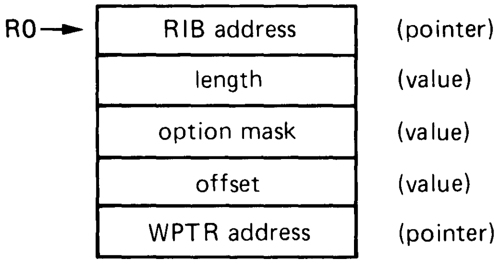
If WD$FIX is specified in the call, however, the content of location wptr prior to the call must be a virtual address in the range of the first or only APR to be modified by the operation. Thus, you force the selection of APRs in fixed mode.

For example, if the precall value in wptr is 140000(octal), corresponding to the base of APR 6, the primitive uses APR 6 and, if needed, APR 7 for the mapping operation, regardless of the free or in-use status of those APRs. Note that only the high-order, APR-selecting portion (the active page field) of the address value in wptr is significant for the operation; MAP$ ignores the displacement field. The virtual address value returned in wptr would be 140000 plus any common-region offset contained in the RIB for the region in question. Normally, the returned address would be exactly 140000 for a physical region or a value between 140000 and 140076 for a shared common region.

If the fixed mode of APR selection is used, Unmap Window calls are not required between successive Map Window calls for iterative remapping of a window.

Argument Block

The calling argument block generated (or assumed to exist) by the MAPW$x macro has the following format:



MLO-444-87

Syntax Example

```
MAPW$S rib=#COMRGN,len=#30000,opt=#WD$FRE,offset=#0,wptr=#WINDOW
```

This stack ($S) form of the macro call requests a window that is one and a half virtual pages in length and that starts at the beginning of the region described by the region ID block COMRGN, as implied by the 0 offset value. Since WD$FRE is specified, the two APRs that are needed for the mapping are to be chosen by the primitive. (The calling process must have two consecutive unused APRs in its current mapping.) The call specifies WINDOW as the location that is to receive the window pointer (the initial virtual address within the mapped window) that is returned by the primitive.

Semantics

In the following description, free APR refers to an APR that is unmapped (whose access control field is set to no access) at the time of the call. Only free APRs are candidates for modification under the WD$FRE option. (An APR that was modified by a prior MAPW$ call can be freed for remapping by an intervening UMAP$ operation.)

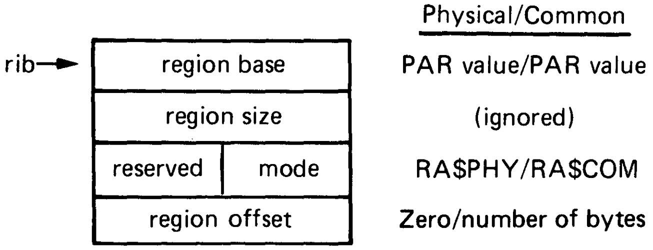
The MAPW$ primitive calculates the number of APRs, n, needed for the window, based on the window length specified in the call plus the region offset, if any, described in the RIB. If I&D-space separation is in effect for the calling process, MAPW$ selects the APR set to be operated on as requested by the WD$INS/WD$DAT option.

If the WD$FIX option was specified in the call, MAPW$ determines the initial or only APR to be mapped, APRi (where i is a value from 0 to 7), from the virtual address value supplied in location wptr. If more than one APR is needed and n APRs do not exist starting with APRi, MAPW$ returns to the caller, with an error indicating "too few APRs available." If the WD$FRE option was specified or defaulted in the call, MAPW$ tests the caller's mapping context for n consecutive free APRs. If n consecutive free APRs are not available, MAPW$ returns to the caller, with an error indicating "too few APRs available." Otherwise, the first of the n free APRs is established as APRi.

MAPW$ then maps the required APRs, modifying both the MMU hardware registers and the corresponding locations in the mapping-context restore area associated with the caller's PCB. MAPW$ forms the physical base address, or PAR value, for APRi by adding the offset specified in the call (in 32-word units, or PAR "ticks") to the region base described in the RIB. PAR values for successive APRs, if any, are increased appropriately. Page descriptor register (PDR) values, specifying access control and page lengths, are set as required. If the WD$DAC option was specified in the call, MAPW$ sets bit 15 of each PDR to disable caching for these APRs.

Finally, MAPW$ forms the window-pointer address by adding the region offset, if any, described in the RIB to the 4K-boundary virtual address that corresponds to APRi and returns that computed value to the wptr location specified in the call.

The information in the user's RIB area must be in the following form:



MLO-448-87

The offset and size symbols defined for the RIB fields are:

RI.ADD	Region base
RI.LEN	Region size
RI.ATR	Region mode (attribute byte)
RI.RES	Reserved (high byte)
RI.OFF	Region offset
RI.SIZ	RIB size in bytes

The RIBDF$ macro in the MicroPower/Pascal COMU and COMM system macro libraries defines these symbols.

The region offset, relevant for a shared common region, is an increment in bytes from the region-base PAR value to the beginning of the region. The region size, which specifies the number of PAR ticks (units of 32 words) contained in the region, is not used in the Map Window operation, since the len parameter of the call determines the length of the mapped window.

The region mode, indicated by the value of the symbol RA$PHY or RA$COM, is not tested by the primitive. Therefore, the offset field must contain 0 for a physical region, as supplied by the ALRG$ or ACSR$ primitive. (The RA$PHY and RA$COM mode symbols are defined by the RIBDF$ macro.)

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD	Invalid address; the RIB address is not on a word boundary.
ES$IPR	Illegal primitive call; the primitive service was requested in an unmapped environment.
ES$NFA	No free APR; insufficient number of APRs available for the requested operation (see Semantics).

Implementation Notes

Since the MMU-register modifications that MAPW$ and UMAP$ perform are reflected by corresponding changes in the caller's mapping-context restore area in one logically indivisible operation, MMU-context saving is not required each time the process is switched out of run state. Such context saving is needed by processes that modify their mapping directly by means of access to the I/O page, at some cost in overall performance. (MMU-context saving is a process-creation option.) This aspect of MAPW$ usage versus self-modification should be weighed in the design of driver, privileged, and device-access processes that require dynamic mapping alterations. Note that the SCTX$ and RCTX$ primitives facilitate saving and restoring of initial mapping values around temporary remappings.

3.32 PDAT$ (Define a Pure-Data Program Section)

Pascal equivalent: None

The PDAT$ macro declares a p-sect of pure data within a MicroPower/Pascal source module. A p-sect declared with PDAT$ has the read-only, data, relocatable, and concatenated attributes; has the name .MCON.; and may be allocated in ROM.

The PDAT$ macro is not a primitive call. Rather, it is an assembly-time macro that is used with the DFSPC$, PURE$, and IMPUR$ macros to segregate read-only code (ROM or RAM), read-only data (ROM or RAM), and read/write data (RAM only) p-sects. During system building, p-sects that have been declared with the DFSPC$, PURE$, PDAT$, and IMPUR$ macros (or by other means) are grouped according to their read-only versus read/write attribute and consolidated by p-sect name by the RELOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.) If I&D-space separation is requested for the static process at build time, RELOC also groups p-sects according to their instruction versus data attribute, so use of both PURE$ and PDAT$ (or their equivalents) is required for that case, as well as IMPUR$. That is, pure data must be segregated from pure code as well as from impure data if I&D separation will be used for the process in question. Although the use of PURE$, PDAT$, and IMPUR$ is not mandatory (if equivalent program sectioning is achieved by other means), these macros are convenient, and their use is recommended.

Program sectioning is implicit in Pascal source programs. That is, appropriate program sectioning is provided transparently by the MicroPower/Pascal compiler.

Syntax

The syntax of the PDAT$ macro call is:

```
PDAT$
```

Semantics

At assembly time, the PDAT$ macro generates a .MCON. p-sect definition with the attributes RO (read-only), D (data), REL (relocatable), CON (concatenated), and LCL (local). The GBL/LCL attribute distinction is not significant for MicroPower/Pascal applications.

Error Returns

Not applicable; this macro is not executable.

3.33 PELC$ (Conditional Put Element)

Pascal equivalent: COND_PUT_ELEMENT Function

The Conditional Put Element (PELC$) primitive implements a nonblocking form of Put Element operation; compare with the unconditional PELM$ primitive. PELC$ 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, because of insufficient space in the buffer. The input-access mode (record or stream) of the ring buffer determines whether the primitive attempts to satisfy the Put request by a full or by a partial transfer, as described below. Informally, the meaning of PELC$ request for a record-mode buffer is "put n bytes into the buffer right away or none at all," and the meaning for a stream-mode buffer is "put as many bytes as will fit, up to n."

In either case, however, PELC$ 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 bytes specified in the call have been successfully put in 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 CRST$ primitive. For a ring buffer with record-mode input, PELC$ attempts to satisfy the request with a full transfer only. If the ring buffer cannot immediately accommodate all bytes to be Put, the primitive returns to the caller, with an R0 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, PELC$ attempts to satisfy the request with either a full or a partial transfer. That is, the primitive puts all bytes that can be immediately 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 PELM$ request to be satisfied, no space is available.

The complementary GELM$, GELC$, and GELA$ primitives allow a process to extract an element from a ring buffer, freeing the corresponding space.

Syntax

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

Variant	Macro Call
PELC$	PELC$ [area,sdb,bufptr,bufcnt]
PELC$S	PELC$S [sdb,bufptr,bufcnt]
PELC$P	PELC$P [sdb,bufptr,bufcnt]

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 data to be transferred to the ring buffer. This argument has the form:

```
[BUFPTR=]buffer-address
```

bufcnt

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

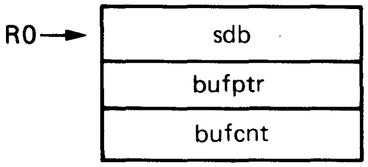
```
[BUFCNT=]integer
```

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. (If it does, the Put request will always fail, since the buffer will never have room for a full record.)

Argument Block

The calling argument block generated (or assumed to exist) by the PELC$x macros has the following format:



MLO–449–87

Syntax Example

```
PELC$ area=#ARGBLK,bufptr=#MSG,bufcnt=#MSGLEN
```

Semantics

If the specified ring buffer's input-access attribute is SA$RIR (record mode), the PELC$ 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 0 in R0. If less than bufcnt bytes of space is available, PELC$ returns immediately with the value bufcnt in R0, indicating that no bytes were Put.

If the specified ring buffer's input-access attribute is SA$RIS (stream mode), PELC$ 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 R0.

A successful PELC$ operation may cause preemption of the caller if the operation unblocks a process waiting to get elements (see Section 2.1.4). That is, return from a successful PELC$ request is not necessarily immediate.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

3.34 PELM$ (Put Element)

Pascal equivalent: PUT_ELEMENT Procedure

The Put Element (PELM$) 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.

In general, if two or more processes are putting data into the same ring buffer, the calling process will block if another process is already waiting for its Put request to be satisfied. The calling process must wait its proper turn (whether by FIFO or priority order) for access to the buffer, since sequential access to a ring buffer is ensured for multiple writers as well as for multiple readers. The process that blocks first is given active write access to the buffer and is never displaced by another, higher-priority process, regardless or the ordering attribute of the waiting-input-process list.

If the ring buffer's input-access mode is stream, the data transfer may occur in increments while the process is waiting. Essentially, stream-mode access permits a ring buffer to be smaller than the "records" that may be passed through it.

The complementary GELM$, GELC$, and GELA$ primitives allow a process to extract an element from a ring buffer, freeing the corresponding space.

The conditional, nonblocking form of PELM$ is the PELC$ primitive.

Syntax

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

Variant	Macro Call
PELM$	PELM$ [area,sdb,bufptr,bufcnt]
PELM$S	PELM$S [sdb,bufptr,bufcnt]
PELM$P	PELM$P [sdb,bufptr,bufcnt]

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 the data to be transferred to the ring buffer. This argument has the form:

```
[BUFPTR=]buffer-address
```

bufcnt

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

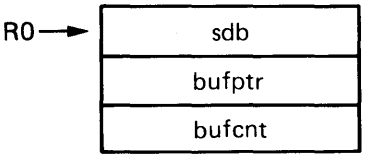
```
[BUFCNT=]integer
```

Restrictions

If the ring buffer's input-access mode is record, the number of bytes to be put must not exceed the size of the ring buffer.

Argument Block

The calling argument block generated (or assumed to exist) by the PELM$x macro has the following format:



MLO–450–87

Syntax Example

```
PELM$S sdb=#OUTRNG,bufptr=#INFO,bufcnt=R1
```

Semantics

If no other process is waiting to put data into the specified ring buffer, the PELM$ 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 the space for the entire transfer is insufficient and the input access mode is record, the primitive blocks the caller with active write access to the ring buffer and calls the scheduler. When sufficient space becomes available as a result of one or more subsequent Get Element operations, the transfer is performed, and the waiting process is unblocked.

If the space for the entire transfer is insufficient and the input access mode is stream, the primitive blocks the caller with active write access to the ring buffer, 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 (possibly by a series of partial transfers) and the waiting process is unblocked.

Note

> In stream mode, partial transfers to a ring buffer (incremental Put operations) can occur whenever three-quarters or more of the buffer is empty, up to the point where the full Put request can be satisfied. Conversely, partial transfers from a ring buffer (incremental Gets) can occur in units of at least one-quarter of the buffer size, up to the point where the full Get request can be satisfied.

If one or more processes are already waiting to put data into the ring buffer at the time of the call, implying that another process then has active write access, the calling process is blocked on the buffer's waiting-input-process list in either FIFO or priority order, depending on the ring buffer definition. (A process with active access is never displaced by another process, regardless of relative priorities.) The process waits its turn to gain active write access, at which point it is treated as described above.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

ES$IPM Illegal parameter; the bufcnt value exceeds the size of the ring buffer for a record-mode operation.

3.35 PURE$ (Define a Pure-Code Program Section)

Pascal equivalent: None

The PURE$ macro declares a program section of pure code within a MicroPower/Pascal source module. A program section declared with PURE$ has the read-only, instruction, relocatable, and concatenated attributes; has the name .MCOD.; and may be allocated in ROM. (A program section for read-only data can be declared with the PDAT$ macro, described elsewhere in this chapter.)

The PURE$ macro is not a primitive call. Rather, it is an assembly-time macro that is used with the DFSPC$, PDAT$, and IMPUR$ macros to segregate read-only code (ROM or RAM), read-only data (ROM or RAM), and read/write data (RAM only) program sections. During system building, program sections that have been declared with the DFSPC$, PURE$, PDAT$, and IMPUR$ macros (or by other means) are grouped according to their read-only versus read/write attribute and consolidated by p-sect name by the RELOC utility. (See Section 2.1.6 for more information on program sectioning under MicroPower/Pascal.) If instruction- and data-space separation is requested for the static process at build time, RELOC also groups p-sects according to their instruction versus data attribute, so use of PDAT$ versus PURE$ (or their equivalents) is required for that case, as well as IMPUR$. Although the use of PURE$, PDAT$, and IMPUR$ is not mandatory (if equivalent program sectioning is achieved by other means), these macros are convenient, and their use is recommended.

Program sectioning is implicit in Pascal source programs. That is, appropriate 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 PDAT$ macro generates a .MCOD. p-sect definition with the attributes RO (read-only), I (instruction), REL (relocatable), CON (concatenated), and LCL (local). The GBL/LCL attribute distinction is not significant for MicroPower/Pascal applications.

Error Returns

Not applicable; this macro is not executable.

3.36 PWFL$ (Powerfail Detection)

Pascal equivalent: POWER_FAIL Function

The Powerfail Detection (PWFL$) primitive allows the caller to determine whether the latest system start-up was a "warm" restart following a power failure or was a cold start. (The warm-restart capability applies only to a target system that has some nonvolatile RAM, as described below.) The PWFL$ primitive returns the kernel-defined value TRUE in R0 if a warm restart has occurred or the value FALSE if the current start was cold, implying that all of read/write memory has been cleared by the kernel's initialization routine (as is always the case for an initial system start-up). The primitive is intended for use in the initialization code of a static process that implements some form of power-fail recovery through use of checkpointing techniques and nonvolatile RAM.

A warm restart following a power failure differs from a cold start or "cold restart" only to the extent that any nonvolatile RAM allocated to a process's impure-data segment is not reinitialized by the kernel during the restart. (Thus, some valid user data may be preserved across the power failure and subsequent power-up, although all kernel data structures are lost and all static processes restarted "from scratch.") Warm restarts are possible only under the following conditions:

- Some or all of the target RAM is declared as nonvolatile in the MEMORY configuration macros (volatile=NO) and is implemented with battery backup. See Section 4.3.7. (If you are debugging under PASDBG and only simulating power failures for testing purposes, the RAM in question need not be nonvolatile in actuality but must be declared as such. Section 4.3.7 provides special debugging information concerning such simulation.)
- All code and pure data resides in nonvolatile memory, whether RAM or ROM.
- The kernel's impure-data area resides in nonvolatile RAM so that the restart indicators are preserved across the power failure, although the area is entirely reinitialized on any restart.
- The impure-data area of any process owning data involved in power-fail recovery resides in nonvolatile RAM.

The PWFL$ primitive invariably returns a FALSE (cold start) indication if none of the target RAM is declared as nonvolatile in the system configuration file, regardless of actual or simulated power failures. Therefore, the use of PWFL$ is meaningful in a simulated, debugging situation only if at least the first condition listed above is satisfied and is meaningful in actual stand-alone operation only if all the stated conditions are satisfied.

Syntax

The syntax of the PWFL$ macro call is:

```
PWFL$
```

Restrictions

The primitive is meaningless for a target system configured with volatile RAM and is not included in the kernel by the PRIMITIVES configuration macro if the target system is so described in the configuration file (see Chapter 4).

Semantics

The PWFL$ primitive returns to the caller, with the value FALSE in R0 if a bootstrap or the kernel has cleared all of read/write memory during the latest system start-up or restart. Alternatively, the primitive returns with the value TRUE if user-process data segments allocated in nonvolatile RAM have not been cleared during the latest restart. Note that the kernel's restart indicators are not reset by the primitive operation.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Error Returns

None.

3.37 P7SYS$ (Enter Normal ISR State)

Pascal equivalent: None

The Enter Normal ISR State (P7SYS$) service request allows a priority-7 interrupt service routine (ISR) to enter kernel-interrupt state (gain normal ISR context) and lower the CPU priority to a specified level. (The priority-7 dispatching mechanism is extremely fast for minimum latency but imposes severe restrictions on the ISR. An ISR that is dispatched to at hardware priority 7 has virtually no context, and the system-state alterations and stack switching needed for normal ISR operation are not performed during the dispatch.) A priority-7 ISR must enter kernel-interrupt state (become a normal ISR) before altering CPU priority or issuing a FORK$ request, since a priority-7 ISR must gain context before becoming interruptable. In contrast, a normally dispatched ISR may raise and lower CPU priority as required for critical instruction sequences.

Note that the P7SYS$ service is not a primitive operation. Rather, it is a special entry point in the interrupt dispatcher that allows a priority-7 IPR to be "redispatched," essentially, after its most critical code has been executed.

Syntax

The syntax of the P7SYS$ macro call is:

```
P7SYS$  pri
```

pri

The desired CPU priority level; an integer from 0 to 6. (On many target systems, only the values 0, 4, 5, and 6 are valid.) This parameter has the form:

```
[PRI=]integer
```

Restrictions

The service may be requested only by an ISR that is initially dispatched to at priority 7.

When the P7SYS$ request is executed, the contents of all registers must be the same as they were on entry to the ISR.

Syntax Example

```
P7SYS$ 4        ; Note: P7SYS$ #4 would be invalid!
```

Semantics

On return from the P7SYS$ request, at the instruction following the call, the ISR is running on the kernel-interrupt stack with normal ISR context and at the specified hardware priority. The ISR can subsequently fork for execution of primitives and/or exit with an RTS PC instruction.

Error Returns

None.

3.38 RBUF$ (Reset Ring Buffer)

Pascal equivalent: RESET_RING_BUFFER Procedure

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

The RBUF$ request is like a GELM$ 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 GELM$ 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 GELM$ request.

Note also that the RBUF$ 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 RBUF$ macro and their respective macro calls are listed below. The differences are described in Section 3.1.

Variant	Macro Call
RBUF$	RBUF$ [area,sdb]
RBUF$S	RBUF$S [sdb]
RBUF$P	RBUF$P [sdb]

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
```

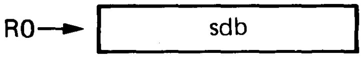
sdb

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 generated (or assumed to exist) by the RBUF$x macro has the following format:



MLO-451-87

Syntax Example

```
RBUF$S sdb=#TTRING
```

Semantics

If no other process is waiting to get bytes from the specified ring buffer, the RBUF$ 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, RBUF$ places the calling process on the buffer's waiting output-process list, as described for a GELM$ 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

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such ring buffer exists. (This error return could be caused by an invalid SDB address.)

3.39 RCTX$ (Restore Context)

Pascal equivalent: RESTORE_CONTEXT Procedure

The Restore Context (RCTX$) primitive permits a process to reset itself to an earlier state of virtual-to-physical mapping previously saved by means of the SCTX$ primitive. RCTX$ restores the APR values that were most recently saved by SCTX$ and updates the mapping-context restore area associated with the caller's PCB accordingly. (The mapping-context restore area contains the process's current mapping image and is used automatically by the kernel during process context switches.)

Used with the SCTX$ primitive, RCTX$ allows a process to reset its entire mapping to a known state. That cancels the effect of intervening alterations of its mapping, especially if such mapping operations involved "borrowing" of one or more statically mapped IPRs.

Multiple calls to RCTX$ without intervening SCTX$ calls cause successively older "generations" of mapping context to be restored, assuming that multiple save operations were executed before the sequence of RCTX$ calls. Multiple copies of mapping context are saved in LIFO order, as described for the SCTX$ primitive. Thus, a process could take "snapshots" of its mapping at several points and then restore the last-saved mapping, the next-to-last, and so on, by a corresponding number of RCTX$ calls.

Chapter 5 contains a general discussion of dynamic mapping.

Syntax

The RCTX$ macro has no variants or arguments; its syntax is:

```
RCTX$
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

No argument block is generated by the RCTX$ macro.

Syntax Example

```
RCTX$
```

Semantics

The RCTX$ primitive copies the mapping-register image contained in the first or only context descriptor block pointed to by the caller's PCB into both the MMU registers and the mapping-context restore area used for process context switching. The primitive then removes the block from the caller's context-descriptor list, deallocates the block, and returns to the caller.

If the caller's context descriptor list is empty, the primitive returns an error indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IPR	Illegal primitive call; no mapping context currently saved by means of SCTX$, or the primitive service was requested in an unmapped environment.

Implementation Note

Like MAPW$ and UMAP$, the RCTX$ primitive alters both the MMU hardware registers and the caller's automatic mapping-context restore area in one logically indivisible operation. Thus, if all mapping alterations are done exclusively through MAPW$ and RCTX$ operations, MMU-context saving is not required each time the process is switched out of run state. Such context saving is needed by a process that modifies its mapping directly through access to the I/O page, at some cost in overall performance: 16 or 32 extra MOV instructions and a few others on every switch from run state. (MMU-context saving is a process-creation option.) This aspect of MAPW$/RCTX$ usage versus self-modification should be weighed in the design of driver, privileged, and device-access processes that require dynamic mapping alterations.

3.40 RCVA$ (Receive Any Data)

Pascal equivalent: $\left\{ \begin{array}{l} \text{RECEIVE\_ANY Procedure} \\ \text{RECEIVE\_ANY\_ACK Procedure} \end{array} \right\}$

The Receive Any Data (RCVA$) primitive implements a complex form of the Receive Data operation; see the RCVD$ and SEND$ primitives for a description of the basic Receive and Send Message operations on queue semaphores. RCVA$ performs the basic Receive operation on the logical OR of several queue semaphores, with an optional timeout feature. That is, RCVA$ permits the calling process to test for and, if necessary, wait on message data on any one of a set of queue semaphores. Up to four queue semaphores may be specified in the primitive request. If no message packet is available on any of the specified semaphores, the calling process blocks until any one of those semaphores is signaled (or sent to) and provides a message for the calling process. (The caller could be blocked behind other waiting processes on a given queue semaphore, of course, although a multiple-receiver policy is unlikely, particularly in the case of complex-primitive usage.) The caller receives message data by value or by reference or by a combination of both, as described for the basic Receive operation.

Optionally, the Receive Any Data operation can be terminated because of the expiration of a time interval specified in the request. On successful return from the operation (C bit clear), R0 will contain either an ordinal value identifying the semaphore that satisfied the request or a 0, indicating that the request timed out.

Thus, the RCVA$ primitive allows a process to get a message from any of up to four queue semaphores, each semaphore being signaled (put or sent to) by a separate process, for example. The primitive might also be used primarily for its timeout capability, regardless of the number of packet queues involved.

If a zero time period (immediate timeout) is specified in the request, the RCVA$ primitive provides a complex form of the Conditional Receive Data (RCVC$) operation, which tests for an available message but will not block the caller. See the RCVC$ primitive for a description of the basic Conditional Receive Data operation.

Syntax

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

Variant	Macro Call
RCVA$	RCVA$ [area,time,rtnptr,vlen,vbuf,rlen,rbuf,sdb-list]
RCVA$S	RCVA$S [time,rtnptr,vlen,vbuf,rlen,rbuf,sdb-list]
RCVA$P	RCVA$P [time,rtnptr,vlen,vbuf,rlen,rbuf,sdb-list]

In each of the macro-call variants, the syntax element shown as sdb-list has the form:

```
sdb1[,sdb2,sdb3,sdb4]
```

area

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

```
[AREA=]arg-blk-address
```

time

The address of a 2-word user-memory location that specifies a timeout interval, expressed in milliseconds. The first word of the double-precision integer contains the low-order portion of the time value, and the second word (time+2) contains the high-order portion. An argument value of 0 implies no timeout for the request; the calling process may block indefinitely. This argument has the form:

```
[TIME=]word-address or #0
```

If the address value is nonzero but the time value pointed to is 0, the request will be timed out immediately if no packet is available on any of the specified semaphores when the primitive is called. That is, the calling process will never block if the specified time interval is 0.

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 in the RCVD$ primitive description. 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 argument value can range from 0 to 34. The 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 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.

sdb-i

The address of a structure descriptor block (SDB) that identifies one of the semaphores to be operated on. From one to four SDB addresses may be specified. The order in which the SDBs are specified (or are identified if enumerated by keyword) determines the order in which the corresponding semaphores are initially tested for a message packet. (That order can be critical under certain real-time conditions, as discussed under Implementation Notes.) The sdb-i arguments have the form:

```
[SDBi=]sdb-address
```

Where i may have the value 1 through 4 if the keyword form of argument is used.

Restrictions

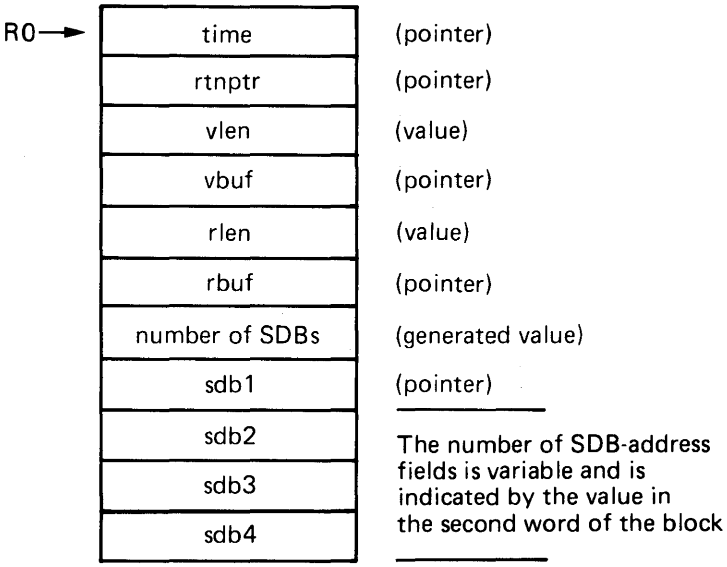
The value of the rlen parameter may not exceed 8129.

The timeout value may not exceed (2**31)–1, the largest positive integer expressible in 32 bits. That is, the sign bit of the time-interval doubleword (bit 15 of the high-order word) must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEP$ primitive for more detail.)

If the keyword form of macro call is used, higher-numbered sdb-i keywords may not be used unless each of the lower-numbered sdb-i keywords is specified. That is, if the keyword sequence contains SDB3=, for example, the sequence must also include SDB1= and SDB2=, although not necessarily in numeric order.

Argument Block

The calling argument block generated (or assumed to exist) by the RCVA$x macro has the following format:



R0→	time	(pointer)
	rtnptr	(pointer)
	vlen	(value)
	vbuf	(pointer)
	rlen	(value)
	rbuf	(pointer)
	number of SDBs	(generated value)
	sdb1	(pointer)
	sdb2	The number of SDB-address fields is variable and is indicated by the value in the second word of the block
	sdb3	
	sdb4	

MLO–452–87

Syntax Example

```
RCVA$P NOWAIT,INFO,40,MSGBUF,0,0,REQSTA,REQSTB,REQSTC,REQSTD
```

This example shows the parameters-only version of the call with a zero reference-length value, a zero (effectively null) reference buffer argument, and four SDB addresses. Up to 40(octal) bytes of data are expected by value, as specified by the vlen parameter. The argument block is generated at assembly time, can be in read-only memory, and can be referred to with the "RCVA$ area" form of call.

Semantics

The RCVA$ primitive tests each semaphore specified in the request for an available queue element, or message packet. (The semaphores are tested in the order in which they are identified in the call, by either position or keyword value.) If any of the semaphores has a packet at the time of the call, the primitive performs a basic Receive operation on the first such semaphore encountered, copying message data to user space as requested, and returns to the caller, with a nonzero value in R0. (The packet pointer is dequeued and returned to the free pool as part of the operation.) The R0 value, an integer between 1 and 4, indicates that the nth semaphore identified in the call satisfied the request.

If none of the semaphores has a packet and either no time argument or a nonzero timer value was supplied in the call, the primitive switches the calling process to the wait-active state. In that state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores has a packet and a zero timer value was supplied in the call, the primitive returns immediately to the caller, with a zero value in R0, indicating a return caused by timeout. (The calling process thus never leaves the run state in the case of an immediate-timeout form of request.)

If the calling process switches to wait-active state, the process is blocked from execution until it can be reactivated either by a packet becoming available on one of the blocking semaphores (see SGLQ$ or SEND$ semantics) or by elapse of the specified timeout period, if any. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or the run state, depending on relative process priorities. If unblocked because of an available packet, R0 contains the ordinal value (from 1 to 4) of the semaphore that provided the message, as described above. If unblocked because of a timeout, R0 contains a 0. In either case, the user's C bit is cleared, distinguishing the value returned in R0 from an error-return indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the timer-value, buffer, rtnptr, or SDB address is not on a word boundary or not in the user's address space. (The addresses are checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure description (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

ES$IPM Illegal parameter; timer value is out of range.

Implementation Notes

Since the initial test of the semaphores for an available packet is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, if the relative frequency of signals or sends is high for one of several queue semaphores and the "fast" semaphore is identified as being first, either by position in the SDB sequence or by the keyword SDB1=, messages on that semaphore will tend to mask off the others in a sequence of RCVA$ operations. Thus, the "slower" semaphores may seldom or never be tested and serviced. Optimally, then, the semaphore with the highest expected signal/send rate should be identified as last, the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so that at least n semaphores are guaranteed to be tested in n calls to RCVA$. The correct or best strategy is application specific.

3.41 RCVC$ (Conditional Receive Data)

Pascal equivalent: $\left\{ \begin{array}{l} \text{COND\_RECEIVE Function} \\ \text{COND\_RECEIVE\_ACK Function} \end{array} \right\}$

The Conditional Receive Data (RCVC$) primitive implements the nonblocking form of the RCVD$ operation. RCVC$ tests a specified queue semaphore for an available packet. If one is available, RCVC$ 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 RCVD$. If the Receive operation is performed successfully, the primitive returns the kernel-defined value TRUE in R0. If not, the kernel-defined value FALSE is returned in R0.

This primitive is the analog of WAQC$ for use by general and device-access processes, which cannot access a packet directly in a mapped environment. RCVC$ permits any type of process to conditionally receive data from another process through a packet. The complementary SEND$ primitives permit any type of process to send data through a packet. See the SGLQ$ primitive for more information about packets.

The message-reception features of RCVC$ are identical to those provided by RCVD$; the copying of messages sent either by value or by reference. The only functional difference between the two primitives is the unconditional Wait performed by RCVD$ versus the conditional Wait performed by RCVC$.

Syntax

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

Variant	Macro Call
RCVC$	RCVC$ [area,sdb,rtnptr,vlen,vbuf,rlen,rbuf]
RCVC$S	RCVC$S [sdb,rtnptr,vlen,vbuf,rlen,rbuf]
RCVC$P	RCVC$P [sdb,rtnptr,vlen,vbuf,rlen,rbuf]

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 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
```

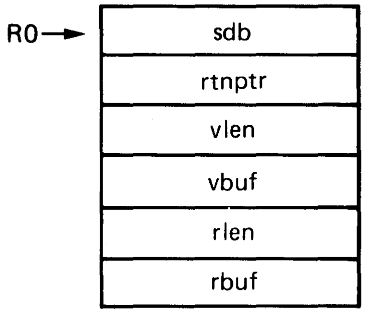
This argument is significant only if rlen is nonzero.

Restrictions

The value of parameter rlen may not exceed 8128.

Argument Block

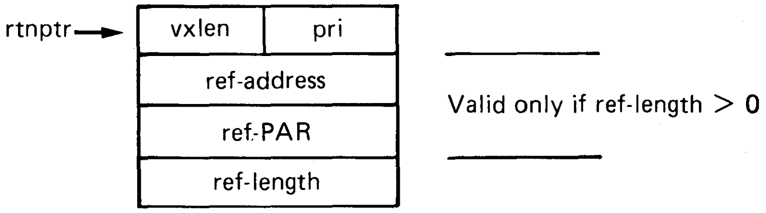
The calling argument block generated (or assumed to exist) by the RCVC$x macro has the following format:



MLO-453-87

Format of Information Returned

The information returned to the caller in the 4-word area pointed to by rtnptr is in the following form:



MLO-454-87

pri

The priority value that was assigned to the packet by the Send operation.

vxlen

The number of bytes sent by value. (This value can be greater than the number of bytes received, which is limited by the vlen argument.) If the value is 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; meaningful only in a mapped environment. 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. (This value can be greater than the number of bytes received, which is limited by the rlen argument.) If the value is 0, no data was sent by reference.

Syntax Example

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

Semantics

The RCVC$ 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:

- 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.
- 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.
- 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.
- Copies the message reference, if any, 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.
- Deallocates the packet, returning it to the kernel's free-element pool for reuse.

RCVC$ then returns control to the caller, with the value TRUE in R0.

If no packet is queued on the specified semaphore at the time of the call, RCVC$ returns immediately to the caller, with the value FALSE in R0, indicating that the Conditional Receive Data operation was unsuccessful. The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Figure 3–1 shows the packet format expected by RCVC$.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer, rtnptr, or SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

3.42 RCVD$ (Receive Data)

Pascal equivalent: { RECEIVE Procedure
RECEIVE_ACK Procedure }

The Receive Data (RCVD$) primitive, the complement of the SEND$ and SNDC$ primitives, permits any type of process to receive data sent through a queue packet. This primitive is the analog of WAIQ$ for use by general and device-access processes, which cannot access a packet directly in a mapped environment.

RCVD$ performs the same Wait operation on a specifed queue semaphore as performed by WAIQ$ but also copies the data in the packet to the caller's buffer space instead of returning the packet pointer. The packet format expected by RCVD$ is the same as that produced by SEND$ or SNDC$ (see Figure 3–1). After the data is copied, the packet is returned to the kernel's free-element pool for reuse. See the SGLQ$ primitive for more information about packets.

A message sent by value (up to 34 bytes) is copied by the RCVD$ 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. (That might be desirable in an unmapped system or if the receiving process is capable of mapping itself to the sender's buffer.)

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 I/O operations, for example, by locking out high-priority driver processes and by delaying the execution of fork routines.

See RCVC$ for the conditional (nonblocking) form of the Receive operation.

Syntax

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

Variant	Macro Call
RCVD$	RCVD$ [area,sdb,rtnptr,vlen,vbuf,rlen,rbuf]
RCVD$S	RCVD$S [sdb,rtnptr,vlen,vbuf,rlen,rbuf]
RCVD$P	RCVD$P [sdb,rtnptr,vlen,vbuf,rlen,rbuf]

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 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
```

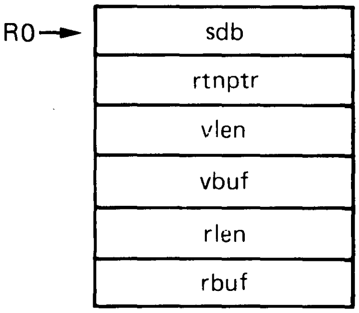
This argument is significant only if rlen is nonzero.

Restrictions

The value of parameter rlen may not exceed 8128.

Argument Block

The calling argument block generated (or assumed to exist) by the RCVD$x macros has the following format:



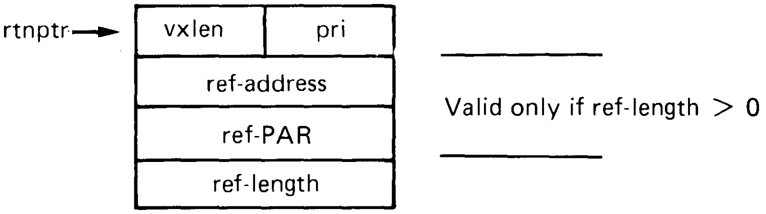
R0 →

sdb
rtnptr
vlen
vbuf
rlen
rbuf

MLO-455-87

Format of Information Returned

The information returned to the caller in the 4-word area pointed to by rtnptr is in the following form:



rtnptr →

vxlen	pri
ref-address	
ref-PAR	
ref-length	

Valid only if ref-length > 0

MLO-456-87

pri

The priority value that was assigned to the packet by the Send operation.

vxlen

The number of bytes sent by value. (This value may be greater than the number of bytes received, which is limited by the vlen argument.) If the value is 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; meaningful only in a mapped environment. 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. (This value may be greater than the number of bytes received, which is limited by the rlen argument.) If the value is 0, no data was sent by reference.

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:

- 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.
- 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.
- 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.
- Copies the message reference, if any, 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.
- Deallocates the packet, returning it to the kernel's free-element pool for reuse.

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

Figure 3–1 shows the packet format expected by RCVD$.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer, rtnptr, or SDB address is not on a word boundary or is not in the user's address space. (The addresses are checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

3.43 REXC$ (Report Exception)

Pascal equivalent: REPORT Procedure

The Report Exception (REXC$) primitive is used to raise a software exception or to simulate a hardware exception. Any type of exception can be reported with this primitive. The caller supplies arguments that the primitive uses in constructing an exception stack frame.

The effect of raising an exception varies; the calling process may be switched to exception-wait state, may execute its own exception service routine, if any, or may be forced to terminate. See Chapter 6 for a description of exception dispatching and the exception stack frame generated by REXC$.

The related SERA$ primitive allows a process to declare an exception service routine.

Syntax

The three variants of the REXC$ macro and their respective macro calls are listed below.

Variant	Macro Call
REXC$	REXC$ [area,mask,code,arglen,argbuf]
REXC$S	REXC$S [mask,code,arglen,argbuf]
REXC$P	REXC$P [mask,code,arglen,argbuf]

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 to be raised, as indicated by a predefined bit-mask symbol or appropriate mask value. The exception-type symbols, of the form EX$xxx, are described in Section 6.1. The symbols are defined by the EXMSK$ macro. This argument has the form:

```
[MASK=]type-symbol
```

code

The specific exception code within the specified type, as indicated by a predefined symbol. The exception-code symbols, of the form ES$xxx, are described in Section 6.1. The symbols are defined by the EXMSK$ macro. This argument has the form:

```
[SUBCOD=]code-symbol
```

arglen

An integer specifying the length in bytes of the optional exception-argument buffer, if any, or 0 for no optional arguments. This argument has the form:

```
[ARGLEN=]byte-count or #0
```

argbuf

The address of the user's exception-argument buffer, if any. The argument value is significant only if arglen is nonzero. This argument has the form:

```
[ARGBUF=]buffer-address or #0
```

Note

In MicroPower/Pascal V2.0 and later, the positional keyword for the exception code argument has been changed from "subcod" to "code." The older form of REXC$x call, specifying "subcod," is upward compatible, however, and will assemble properly.

Restrictions

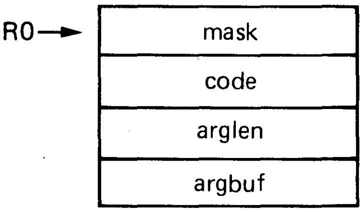
The exception argument buffer, if any, must begin on a word boundary, and the buffer length must be an even number of bytes. (The argument buffer contents are pushed on the stack as the optional part of the exception stack frame, shown in Section 6.5. Thus, the arguments, if any, must be word oriented.)

Exception Type/Code Symbols and Values

See Table 6–1 for a complete listing of exception type and code symbols. See Section 6.2.1 for a program fragment that shows how to derive an exception-type mask value from an exception code.

Argument Block

The calling argument block generated (or assumed to exist) by the REXC$x macro has the following format:



MLO–457–87

Syntax Example

```
REXC$S mask=#EX$HIO,code=#ES$NXU,arglen=#0,argbuf=#0
```

Semantics

The REXC$ primitive causes the kernel's exception dispatcher to be entered for normal disposition of the exception condition indicated by mask and code. Before transferring control to the dispatcher, the primitive pushes the following items on the reporting process's stack, as part of the exception stack frame:

- The optional arguments from the caller's argument buffer, if any
- The argument byte count (arglen value); may be 0
- The exception code
- The exception type (top of stack frame)

Note that the primitive-dispatch mechanism has already saved standard register context (PS, PC, R5, R4, R3) in the stack frame. Also, the optional argument words are pushed in the order in which they occur in the caller's argument buffer. Thus, they appear in "reverse order" in the stack frame. See Section 6.5 for the exception stack-frame format.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the argument buffer address was not on a word boundary or was not within the process's address space, or the process's stack pointer was corrupt at the time of the call.

ES$IPM Illegal parameter; more than one exception type was specified (two or more bits set in mask), no exception type was specified (no bit set in mask), or the argument buffer length value was odd.

3.44 RSUM$ (Resume Process)

Pascal equivalent: RESUME Procedure

The Resume Process (RSUM$) primitive reactivates a suspended process, assuming that the process's current suspension count is –1, which implies current suspension but no additional suspensions pending. (The primitive increments the subject process's suspension count and returns a TRUE or FALSE indication, as described in the Semantics section below.) This primitive allows the calling process to unsuspend another process (see the SPND$ 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]

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
```

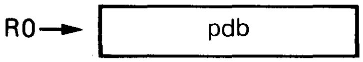
pdb

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:



MLO–458–87

Syntax Example

```
RSUM$S pdb=#APROC
```

Semantics

The RSUM$ 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 ready active; otherwise, the primitive returns to the caller after incrementing the count.

Because of the suspension counter, a Resume Process request may not in fact 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 active. A FALSE return indicates that the process was not changed from the suspended state, because the suspension count remained negative after the increment.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

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

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such process exists. (This error return could be caused by an invalid PDB address.)

3.45 SALL$ (Signal All Waiters)

Pascal equivalent: SIGNAL_ALL Procedure

The Signal All Waiters (SALL$) primitive unblocks the processes waiting on a specified binary or counting semaphore, setting the semaphore variable value to 0. If no process is waiting on the semaphore, the SALL$ operation leaves the value of the semaphore variable unchanged.

This primitive 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 SGLC$, the conditional signal.)

Together, the WAIT$ and SALL$ calls allow two or more processes to synchronize on a single event signaled by another process.

Syntax

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

Variant	Macro Call
SALL$	SALL$ [area,sdb]
SALL$S	SALL$S [sdb]
SALL$P	SALL$P [sdb]

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:

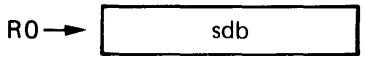
```
[SDB=]sdb-address
```

Restrictions

The semaphore identified by the passed SDB must not be a queue semaphore.

Argument Block

The calling argument block generated (or assumed to exist) by the SALL$x macro has the following format:



MLO-459-87

Syntax Example

```
SALL$S sdb=#SYNCH
```

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

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

3.46 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 another equal-priority process, thus circumventing the kernel's event-triggered scheduling mechanism (see the Applications section).

Syntax

The syntax of SCHD$ macro call 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 action causes the first process in the queue to be switched to the run state.

If no ready-active process is 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 situation 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 the latter a chance to execute.

3.47 SCTX$ (Save Context)

Pascal equivalent: SAVE_CONTEXT Procedure

The Save Context (SCTX$) primitive permits a process to save a copy of its current memory mapping for subsequent restoration by means of the RCTX$ primitive. SCTX$ saves the contents of the calling process's mapping registers (APRs) in a context block that is distinct from the mapping-context restore area always associated with a process's PCB. (The latter area is used implicitly by the kernel during process context switching.)

Used with the RCTX$ primitive, SCTX$ allows a process to reset its entire mapping to a prior, known state, canceling the effect of intervening alterations of its mapping, especially if such mapping operations involved "borrowing" of one or more statically mapped APRs. Typically, SCTX$ might be used preceding a fixed-mode MAPW$ call or preceding an analogous remapping operation performed directly by a privileged or driver-mapped process. Assuming that the remapping is of a temporary nature, an RCTX$ call would be used at some later point to restore the previous mapping.

Successive calls to SCTX$ without intervening RCTX$ calls cause multiple copies of mapping context to be saved in a list structure treated by RCTX$ as a LIFO push-down stack. Thus, a process could take "snapshots" of its mapping at various points and then restore the last-saved mapping, the next-to-last, and so on, by an appropriate number of successive RCTX$ calls.

Together, the SCTX$ and RCTX$ primitives facilitate easy, uncomplicated restoration of mapping at a relatively small cost in performance. Also, if used with MAPW$ (as opposed to direct MMU modification) that set of primitives eliminates the need for MMU-register saving during process context switches from run state, an overall performance benefit.

Chapter 5 contains a general discussion of dynamic mapping.

Syntax

The SCTX$ macro has no variants or arguments; its syntax is:

```
SCTX$
```

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

No argument block is generated by the SCTX$ macro.

Syntax Example

```
SCTX$
```

Semantics

The SCTX$ primitive allocates a context descriptor block in system-common memory and copies the contents of the user's MMU registers into the block. The primitive links the block into the context-descriptor list pointed to by the caller's PCB, as the first or only element of that list, and returns to the caller.

In an LSI–11/73 or similar target environment, the primitive saves both the I&D-space mapping registers if I&D-space separation is in effect for the calling process. Otherwise, only the I-space APR set is saved.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IPR	Illegal primitive call; the primitive service was requested in an unmapped environment.
ES$NMK	Insufficent space for kernel structure; a context descriptor block could not be allocated.

3.48 SEND$ (Send Data)

Pascal equivalent: SEND Procedure

Pascal variant: SEND_ACK Procedure

The Send Data (SEND$) primitive allocates a queue packet in kernel space, copies user data into the packet, and signals a specified queue semaphore. The Signal operation is the same as that performed by SGLQ$. The SEND$ 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 (RCVD$) primitive permits any type of process to receive data sent through a packet. See the SGLQ$ primitive for more 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, if mapped, the buffer's PAR value are placed in the packet for subsequent use by the RCVD$ primitive. The message is transmitted (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 RCVD$ primitive, the receiver can specify that only the message reference is to be returned. (That might be done in an unmapped system or by a receiving process capable of mapping itself to the sender's buffer.) 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 I/O operations (for example, by locking out high-priority driver processes and by delaying the execution of fork routines).

See SNDC$ for the conditional-signal form of the Send Data operation.

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,rlen,rbuf]
SEND$S	SEND$S [sdb,pri,vlen,vbuf,rlen,rbuf]
SEND$P	SEND$P [sdb,pri,vlen,vbuf,rlen,rbuf]

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 CRST$). 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) 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
```

The argument value 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 along with the caller's PAR value that maps the address, forming a 2-word "physical address" in the mapped case. (See Figure 3–1.) This argument has the form:

```
[RBUF=]buffer-address
```

The argument value is significant only if rlen is nonzero.

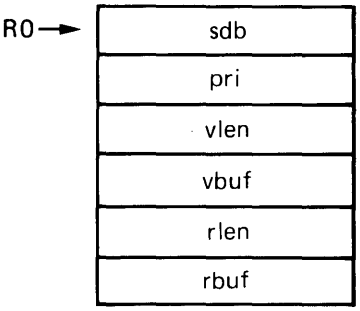
Restrictions

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 8128.

Argument Block

The calling argument block generated (or assumed to exist) by the SEND$x macro has the following format:



MLO-460-87

Syntax Example

```
SEND$ area=#ARGBLK,sdb=#MSGSEM,pri=#0,vlen=#34.,vbuf=#MSG,rlen=#100.,rbuf=#MSG+34.
```

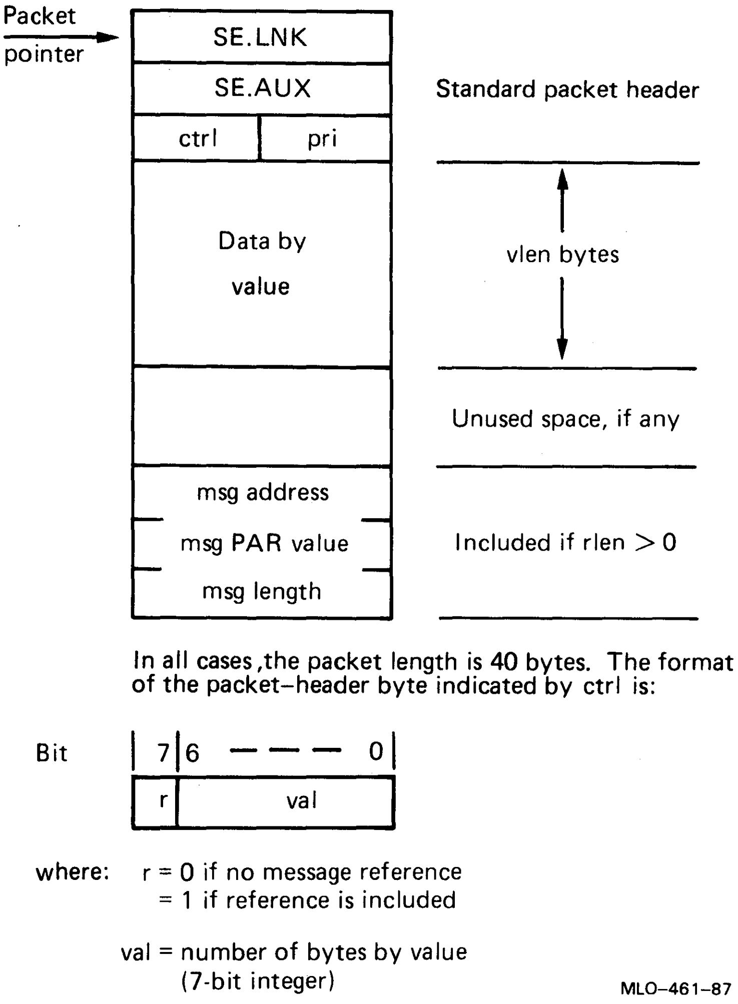
Semantics

The SEND$ primitive performs the following actions before signaling the specified queue semaphore:

1. Obtains a packet (queue element) from the kernel's free-packet 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 RCVD$.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in kernel space.
4. Copies the address (rbuf) of the message to be sent by reference, if any, along with the message length, into the packet. In the mapped case, the virtual address is followed by a corresponding PAR value, forming a 2-word "physical address" in the caller's address space. The value of the first word is the specified virtual address; the value of the second word is the content of the caller's PAR associated with that virtual address. In the unmapped case, of course, the specified address is itself a physical address.

SEND$ then signals the semaphore, placing the packet on the semaphore's queue, as described for the SGLQ$ primitive. Figure 3–1 shows the format of a packet constructed by SEND$ (or by SNDC$).

Figure 3–1: SEND$/SNDC$ Packet Format



In all cases ,the packet length is 40 bytes. The format of the packet–header 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)

Note that synchronization characteristics differ between data sent by value and data sent by reference. That is, data sent by value is copied immediately by SEND$, thus freeing the buffer it occupied for immediate reuse on return from the primitive. Data sent by reference, however, is not copied until the corresponding RCVx$ primitive is executed, and thus the buffer it occupies is not free for reuse by the sender until then. Therefore, the sender and receiver should implement the necessary mutual exclusion of a sender's by-reference buffer through a separate binary semaphore.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The addresses are checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; the amount of data to be sent by value (vlen parameter) exceeds packet capacity, or the amount of data to be sent by reference exceeds 8128 bytes.

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

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, SEND$ is used to implement the interface to higher-level services such as those provided by the communications and device-driver processes. This interface consists of a request message sent to the appropriate system process and a reply received from the process, using the SEND$/RCVD$ facility.

3.49 SERA$ (Set Exception Routine Address)

Pascal equivalents: $\left\{ \begin{array}{l} \text{ESTABLISH Procedure} \\ \text{REVERT Procedure} \end{array} \right\}$

The Set Exception Routine Address (SERA$) primitive establishes an exception service routine within the calling process for a specified set of exceptions. This primitive allows a process to regain control at its exception entry point after causing a particular type of exception, in either of two circumstances:

- No exception-handling process exists for the type of exception and the exception group of the faulting process.
- The existing exception handler chooses to pass the exception on to the faulting process (by means of the DEXC$ primitive).

When the process causing the exception is reentered at its exception routine address, the user's stack contains an exception stack frame 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]

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
```

adr

The address of the exception routine for the process or 0. (See the note at the end of this subsection.) This argument has the form:

```
[ADR=]exception-entry-point or #0
```

mask

The type(s) of exception, as indicated by predefined bit-mask symbols, to be accepted by the exception service routine. The exception type symbols, of the form EX$xxx, are defined by the EXMSK$ macro and are described in Section 6.1. The type symbols may be ORed as desired. This argument has the form:

```
[MASK=]symbol[!symbol]
```

Note

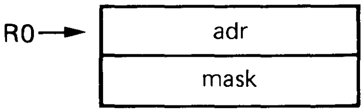
If the adr argument value is 0, the meaning of the request changes to "disable exception address" for the calling process. That is, a call specifying an exception address of 0 cancels any previous SERA$ call made by the process and disables the passing of any exceptions to the process. The mask argument is not meaningful in this case.

Restrictions

This primitive may be used only at process level; it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the SERA$x macro has the following format:



MLO–463–87

Semantics

The SERA$ primitive places the specified exception entry address and exception-type bit mask in the caller's PCB (fields PC.EXC and PC.MSK), replacing the previous values, if any, and returns to the caller.

When the kernel passes an exception back to the process that caused it, the exception service routine is entered as described in Sections 6.3 and 6.4.2.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the exception routine address was an odd value or not in the process's address space.

ES$IPR Illegal primitive; SERA$ was called from an ISR.

3.50 SGLC$ (Conditionally Signal Semaphore)

Pascal equivalent: COND_SIGNAL Function

The Conditionally Signal Semaphore (SGLC$) 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 (its 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 action 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 one of a set of semaphores, unblocking one process, if any, waiting on a semaphore of that set.

Unlike the SGNL$ primitive, the relative order in which the Signal and Wait occur affects the operation of the SGLC$ primitive. The SGLC$ call provides, for example, a means of testing each of a set of identical server processes for availability (see also WAIC$).

Syntax

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

Variant	Macro Call
SGLC$	SGLC$ [area,sdb]
SGLC$S	SGLC$S [sdb]
SGLC$P	SGLC$P [sdb]

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:

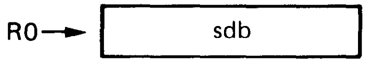
```
[SDB=]sdb-address
```

Restrictions

The semaphore identified by the passed SDB must not be a queue semaphore.

Argument Block

The calling argument block generated (or assumed to exist) by the SGLC$x macro has the following format:



MLO-464-87

Syntax Example

```
SGLC$S R2
```

Semantics

The SGLC$ primitive signals the specified semaphore only if at least one process is waiting. Otherwise, the primitive returns immediately to the caller, with the 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 action may cause the calling process to be preempted (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 value TRUE, indicating that the semaphore was signaled.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

3.51 SGLQ$ (Signal Queue Semaphore)

Pascal equivalent: PUT_PACKET Procedure

The Signal Queue Semaphore (SGLQ$) 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 when that happens.) 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 message packet it needs, or will need, is available, whether or not the other process is presently waiting for the signal. (Compare with SGQC$, the Conditionally Signal Queue Semaphore 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 34(decimal) bytes, given by the global symbol QE.LEN. A process can obtain a "free" packet by means of an Allocate Packet (ALPK$ or ALPC$) request. (The content of a packet as allocated is undefined, that is, not zeroed.) A packet can be returned to the kernel's free-packet pool through use of the DAPK$ primitive.

In a mapped environment, general and device-access processes do not have direct access to a packet, since they are not mapped to kernel data space; they cannot move data into a packet, for example. Therefore, if such a process needs to send data by means of a packet, as opposed to simply "passing along" a packet pointer that it has acquired by means of another primitive operation, it must use the SEND$ primitive. SEND$ is a higher-level primitive that provides a packet-acquisition and data-copying service in addition to the functionality of SGLQ$.

The WAIQ$ call is the inverse of the SGLQ$ call.

Syntax

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

Variant	Macro Call
SGLQ$	SGLQ$ [area,sdb,qelm]
SGLQ$S	SGLQ$S [sdb,qelm]
SGLQ$P	SGLQ$P [sdb,qelm]

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 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
```

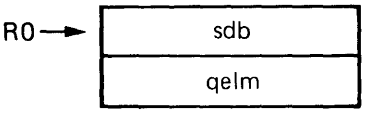
qelm

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
```

Argument Block

The calling argument block generated (or assumed to exist) by the SGLQ$x macro has the following format:



MLO–465–87

Syntax Example

```
SGLQ$S sdb=#QSEM,qelm=#ALPCBF
```

Semantics

The SGLQ$ 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 action may cause the calling process to be preempted, depending on the priority of the unblocked process.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

Applications

Queue semaphores may be used to implement general queuing functions. The SGLQ$ 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.

3.52 SGNL$ (Signal Semaphore)

Pascal equivalent: SIGNAL Procedure

The Signal Semaphore (SGNL$) primitive signals a specified binary or counting semaphore, unblocking the first process, if any, waiting on that semaphore. This primitive permits the calling process to signal to another process that an event has occurred, whether or not the other process is presently waiting for the signal. (Compare with SGLC$, the conditional signal.)

When a binary semaphore acts as a "gate" for mutual exclusion, the Signal Semaphore primitive permits the calling process to open the gate for another process that the caller has previously closed "behind itself" with a WAIT$ call. In this case, both processes issue a WAIT$ before, and a SGNL$ following, a critical section of code; critical relative to the operation of the other process.

Together, the WAIT$ and SGNL$ calls allow two or more cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms (see also SGLC$, WAIC$, and SALL$).

Syntax

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

Variant	Macro Call
SGNL$	SGNL$ [area,sdb]
SGNL$S	SGNL$S [sdb]
SGNL$P	SGNL$P [sdb]

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
```

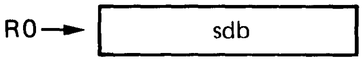
sdb

The address of a structure descriptor block (SDB) that identifies the binary or counting 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
```

Argument Block

The calling argument block generated (or assumed to exist) by the SGNL$x macro has the following format:



MLO–466–87

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 action 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 no process is waiting, the primitive returns immediately to the caller.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

3.53 SGQC$ (Conditionally Signal Queue Semaphore)

Pascal equivalent: COND_PUT_PACKET Function

The Conditionally Signal Queue Semaphore (SGQC$) 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 message packet to another process, but only if the other process is already waiting for the packet. (Compare with SGLQ$, the unconditional Signal Queue call.)

See the SGLQ$ primitive for a description of queue packets. In a mapped environment, general or device-access processes would normally use the SNDC$ primitive instead of SGQC$, since they are not mapped to the kernel data space in which packets reside and thus cannot access a packet directly. The higher-level SNDC$ primitive provides a packet-acquisition and data-copying service in addition to the functionality of SGQC$.

The WAQC$ call is the inverse of the SGQC$ call.

Syntax

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

Variant	Macro Call
SGQC$	SGQC$ [area,sdb,qelm]
SGQC$S	SGQC$S [sdb,qelm]
SGQC$P	SGQC$P [sdb,qelm]

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 queue 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
```

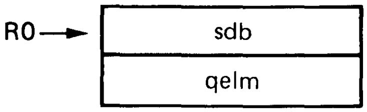
qelm

The address of a pointer to the packet that is to be passed by means of the queue semaphore. This argument has the form:

```
[QELM=]pointer-address
```

Argument Block

The calling argument block generated (or assumed to exist) by the SGQC$x macro has the following format:



MLO-467-87

Syntax Example

```
SGQC$S sdb=#QSEM,qelm=R5
```

Semantics

The SGQC$ primitive tests the specified queue semaphore for a waiting process. 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 operation may cause the calling process to be preempted (lose control of the CPU) depending on the priority of the unblocked process. On eventual return to the caller, R0 contains the value TRUE, indicating that the semaphore was signaled and that the packet was sent. (The caller's packet pointer is no longer valid at that point.)

If no process is waiting on the semaphore, the primitive returns immediately to the caller, with the kernel-defined value FALSE in R0, indicating that the semaphore was not signaled. (The caller's packet pointer is consequently still valid.)

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the SDB address is not on a word boundary or is not in the user's address space. (The address is checked only if the CHECK option is selected in the configuration file.)

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

Applications

This primitive permits a process to 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 SGQC$ call allows the submitting process to test each semaphore for an output process that is ready to service the request.

3.54 SLEP$ (Sleep)

Pascal equivalent: SLEEP Procedure

The Sleep (SLEP$) primitive "puts the calling process to sleep" for at least the period of time specified in the call. More precisely, SLEP$ blocks the calling process in the wait-active state until the closest approximation in clock "ticks" equal to or exceeding the specified time interval has elapsed. At that point, the process changes to the ready-active state, from which it may be switched to the run state, depending on the relative priorities of the awakened process and the current running process. (If the process was suspended during its sleep, it changes to ready suspended rather than ready active, of course.)

The caller specifies the wakeup time as a number of milliseconds following execution of the SLEP$ call. The sleeping process is never woken in less than the specified time. The range of positive difference between the specified and actual wakeup interval is a function of both the clock frequency and relative process priorities. For example, a 60–Hz system clock "ticks" only once every 16.7 milliseconds. See the GTIM$ primitive for more details.

The wakeup interval, specified as a double-precision, 31-bit integer, can range from one millisecond (useful only with an 800–Hz clock) to roughly 24.89 days (see Restrictions for exact value).

The STIM$, GTIM$, and SLEP$ primitives together replace the functionality previously provided by the DIGITAL-supplied clock service process, which is now obsolete.

Syntax

The three variants of the SLEP$ macro and their macro calls are listed below. The differences are described in Section 3.1.

Variant	Macro Call
SLEP$	SLEP$ [area,interv]
SLEP$S	SLEP$S [interv]
SLEP$P	SLEP$P [interv]

area

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

```
[AREA=]arg-blk-address
```

interv

The address of a 2-word area in user memory containing the desired wakeup inverval, as described under Semantics. This argument has the form:

```
[interv=]area-address
```

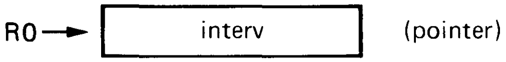
Restrictions

The interv argument must specify an even address.

The interval value is limited to a 31-bit positive integer; that is, the sign bit of the high-order word must not be set. The maximum valid value, (2**31)–1 milliseconds, specifies a timeout period of 24 days, 20 hours, 31 minutes, and 23.647 seconds.

Argument Block

The calling argument block generated (or assumed to exist) by the SLEP$x macro has the following format:



MLO–468–87

Syntax Example

```
SLEP$S interv=#WAKEUP
```

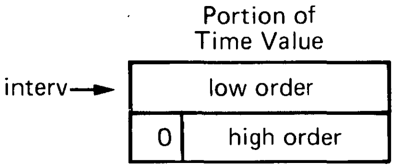
Semantics

The SLEP$ primitive blocks the calling process on the system timer queue, adjusting the queue order and current expiration values as required, and calls the scheduler. The system timer queue is maintained jointly by the primitive and the basic kernel clock-service mechanism. The queue is time ordered; sleeping processes are queued on it in ascending order of wakeup time.

If the interval value supplied in the call is 0, SLEP$ treats the request as a null operation and returns control to the caller.

At each "tick" of the system clock, the kernel's clock interrupt service updates the system time, checks the timer queue, and unblocks any process(es) whose time interval has expired. Each unblocking implies a possible scheduler call.

SLEP$ assumes that the value supplied in the caller's interv area is in the following form:



MLO–469–87

The 0 in the diagram refers to the sign bit of interv+2.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD	Invalid address; the interv address is an odd value.
ES$IPR	Invalid parameter; the interv value exceeds (2**31)–1.

3.55 SNDC$ (Conditional Send Data)

Pascal equivalent: COND_SEND Function

Pascal variant: COND_SEND_ACK Function

The Conditional Send Data (SNDC$) primitive implements a selective form of the SEND$ operation. SNDC$ 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 (1) in R0. If the Send was not performed, the kernel-defined value FALSE (0) is returned in R0. The packet-allocation and data-copying portion of the operation is not done if no process is waiting.

This primitive is the analog of SGQC$ for use by general and device-access processes, which cannot access a packet directly in a mapped environment. SNDC$ permits any type of process to conditionally transmit data to another process through a packet. The complementary RCVD$ primitive permits any type of process to receive the data sent through a packet. See the SGLQ$ primitive for more information about packets.

The message-transmission features of SNDC$ are identical to those provided by SEND$; the sending of messages by value or by reference. The only functional difference between the two primitives is the unconditional signal performed by SEND$ versus the conditional signal performed by SNDC$.

Syntax

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

Variant	Macro Call
SNDC$	SNDC$ [area,sdb,pri,vlen,vbuf,rlen,rbuf]
SNDC$S	SNDC$S [sdb,pri,vlen,vbuf,rlen,rbuf]
SNDC$P	SNDC$P [sdb,pri,vlen,vbuf,rlen,rbuf]

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 CRST$). 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 also 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
```

The argument value 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 along with the caller's PAR value that maps the address, forming a 2-word "physical address" in the mapped case (see Figure 3–1). This argument has the form:

```
[RBUF=]buffer-address
```

The argument value is significant only if vlen is nonzero.

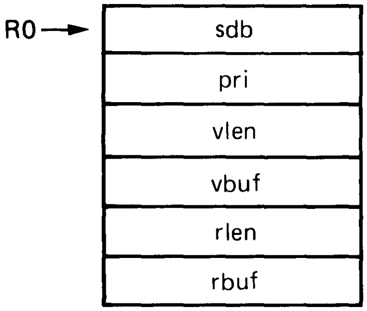
Restrictions

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 8128.

Argument Block

The calling argument block generated (or assumed to exist) by the SEND$x macro has the following format:



R0 →

sdb

pri

vlen

vbuf

rlen

rbuf

MLO-470-87

Syntax Example

```
SNDC$ area=#ARGBLK,sdb=#SERVOR,pri=#10,vlen=#0,vbuf=#0,rlen=#200,rbuf=#LONG
```

Semantics

The SNDC$ primitive tests the specified queue semaphore for a waiting process and performs the following actions before signaling the semaphore:

1. Obtains a packet (queue element) from the kernel's free-packet 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 RCVD$.
3. Copies the data, if any, to be transmitted by value from the buffer in user space to the packet in kernel space.
4. Copies the address (rbuf) of the message to be sent by reference, if any, along with the message length, into the packet. In the mapped case, the virtual address is followed by a corresponding PAR value, forming a 2-word "physical address" in the caller's address space. The value of the first word is the specified virtual address; the value of the second word is the content of the caller's PAR associated with that virtual address. In the unmapped case, of course, the specified address is itself a physical address.

SNDC$ then signals the semaphore, unblocking the first waiting process and passing the packet to it. This action may cause the calling process to be preempted, depending on the priority of the unblocked process (see SGLQ$). On eventual return to the caller, R0 contains the value TRUE, indicating a successful operation.

If no process is waiting on the semaphore, the primitive returns immediately to the caller, with the value FALSE in R0, indicating that the Send operation was not performed.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

A packet constructed by SNDC$ has the same format as one constructed by the SEND$ primitive (see Figure 3–1).

Note

> When using SNDC$, the calling process can be blocked waiting for allocation of a queue element. In this respect, the SNDC$ primitive differs from other conditional primitives, which cannot block the caller. (Thus, this primitive must not be issued by an ISR at fork level; a very unlikely point of use for any higher-level primitive.)

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; buffer or SDB address is not on a word boundary or is not in the user's address space. (The addresses are checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; the amount of data to be sent by value (vlen parameter) exceeds packet capacity, or the amount of data to be sent by reference exceeds 8128 bytes.

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address if address checking is not in effect.)

Applications

The SNDC$ 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.

3.56 SPND$ (Suspend Process)

Pascal equivalent: SUSPEND Procedure

The Suspend Process (SPND$) 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 another process (see the RSUM$ primitive). Together, the SPND$ and RSUM$ primitives provide a mutual- (or unilateral) exclusion mechanism more radical than that provided by the WAIT$ and SGNL$ primitives.

Syntax

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

Variant	Macro Call
SPND$	SPND$ [area,pdb]
SPND$S	SPND$S [pdb]
SPND$P	SPND$P [pdb]

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
```

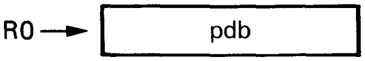
pdb

The address of the process descriptor block (PDB) that identifies the process to be suspended, or 0 to imply the caller. See Section 3.1.6 for the format and use of a PDB. This argument has the form:

```
[PDB=]pdb-address or #0
```

Argument Block

The calling argument block generated (or assumed to exist) by the SPND$x macro has the following format:



MLO-471-87

Syntax Example

```
SPND$S pdb=#0 ;#Suspend self
```

Semantics

The SPND$ primitive decrements the suspension count associated with the specified process (word PC.SPC of the PCB). If the count changes from 0 to –1, the state of the process is changed to either ready suspended or wait suspended, depending on the process's 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.

Because of the possible cumulative effect of multiple Suspend and Resume operations on a suspension counter, a given Suspend request may not immediately suspend 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 (the suspension counter value is –1 or less). A FALSE return indicates that the process was not suspended by the Suspend operation (the suspension counter value is still positive).

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Note that a Suspend operation on a stopped process is always ineffective; see the STPC$ primitive.

A transition from the wait-suspended state to the ready-suspended state can occur while a process is suspended. (An SGNL$ operation can unblock a waiting suspended process, for example.) A Resume operation is required to reactivate a suspended process, however.

The suspension counter is treated as a signed integer value. Thus, the counter can record a maximum of 32,768 successive Suspend requests or 32,767 successive Resume requests.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such process exists. (This error return could be caused by an invalid SDB address.)

3.57 SSFA$ (Set Stop Flag Address)

Pascal equivalent: DEFINE_STOP_FLAG Procedure

The Set Stop Flag Address (SSFA$) primitive effectively allows the calling process to disable the effect of a Stop Process request issued against the caller by another process and to receive instead an indication that such a request has occurred. More specifically, the SSFA$ primitive establishes the address of a stop-flag byte, which the kernel sets to TRUE when another process issues a Stop Process request against the subject process (see the STPC$ primitive).

The SSFA$ primitive also allows the caller to eliminate a previously established stop flag, which effectively reenables the normal, immediate effect of a Stop Process request issued against the caller. Note that the existence of a stop flag for a given process does not inhibit the process from stopping itself with a reflexive STPC$ call, nor does it inhibit an implicit stop (or process abort) resulting from an unhandled exception condition.

The SSFA$ primitive is intended to permit a process to defer execution of its termination routine, in response to Stop Process request, until an appropriate point is reached in its normal execution cycle or to modify its normal execution path before terminating. The subject process can periodically test its stop flag (for example, just before issuing an I/O request) and take appropriate action, depending on the TRUE or FALSE state of the flag. Also, if the flag value were TRUE, the process could gracefully terminate its I/O operations and perform any required signals to other processes before executing a STPC$ on itself.

The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Syntax

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

Variant	Macro Call
SSFA$	SSFA$ [area,addr]
SSFA$S	SSFA$S [addr]
SSFA$P	SSFA$P [addr]

area

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

```
[AREA=]arg-blk-address
```

addr

The address of a byte location in user memory that the kernel will use as a stop flag for the calling process or 0. An address value of 0 requests that use of the caller's existing stop flag is to be discontinued. This argument has the form:

```
[ADDR=]byte-address or #0
```

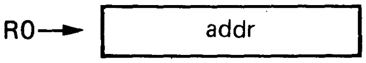
Restrictions

The kernel assumes that the specified byte location is cleared, corresponding to a FALSE state, when the primitive is issued.

The primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the SSFA$x macro has the following format:



MLO–472–87

Syntax Example

```
SSFA$S addr=#STPFLG
```

Semantics

If the address specified in the call is nonzero and, in a mapped envionment, is within the caller's address space, the SSFA$ primitive places that address in field PC.SFA of the caller's PCB and returns. If the address specified in the call is 0, the SSFA$ primitive clears field PC.SFA of the caller's PCB and returns.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; the specified stop-flag location is not within the process's address space (mapped only).

ES$IPR Illegal primitive; the request was issued from an ISR.

3.58 STIM$ (Set Time)

Pascal equivalent: SET_TIME Procedure

The Set Time (STIM$) primitive sets the system time maintained by the kernel to an arbitrary base time value. The caller supplies the base time as a triple-precision integer contained in a 3-word area specified in the call. Presumably, the base time value represents some number of milliseconds. (Use of STIM$ assumes that a system clock is present and configured on the target system.)

The STIM$, GTIM$, and SLEP$ primitives together replace the functionality previously provided by the DIGITAL-supplied clock service process, which is now obsolete.

Syntax

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

Variant	Macro Call
STIM$	STIM$ [area,time]
STIM$S	STIM$S [time]
STIM$P	STIM$P [time]

area

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

```
[AREA=]arg-blk-address
```

time

The address of a 3-word area in user memory containing the value that is to replace the current system time value. This argument has the form:

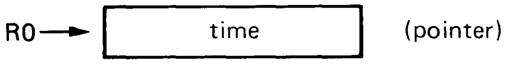
```
[time=]area-address
```

Restrictions

The time argument must specify an even address.

Argument Block

The calling argument block generated (or assumed to exist) by the STIM$x macro has the following format:



MLO–473–87

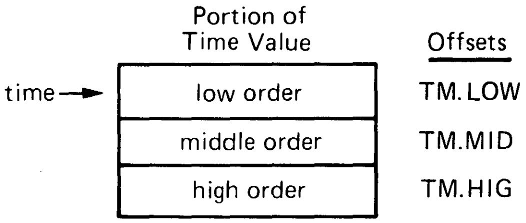
Syntax Example

```
STIM$S time=#DATTIM
```

Semantics

The STIM$ primitive disables interrupts, moves the three words pointed to by the call into the corresponding words of the system-time variable in the kernel's impure area, enables interrupts, and returns to the caller.

STIM$ assumes that the caller's time area has the following form:



MLO–474–87

The TM.xxx offset symbols used by the kernel are defined by the TIMDF$ macro.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IAD Invalid address; the time address is an odd value.

3.59 STPC$ (Stop Process)

Pascal equivalent: STOP Procedure

The Stop Process (STPC$) primitive stops a specified process by forcing it to its termination entry point when it is next reentered (assuming that the subject process does not have a stop flag in effect). 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 STPC$ 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 reentered at its termination point, the stopped process 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 DLPC$ request.

Alternatively, if the subject process has a stop flag in effect (see SSFA$) and is not the caller, the STPC$ primitive sets the subject process's stop flag to TRUE and has no additional effect.

Syntax

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

Variant	Macro Call
STPC$	STPC$ [area,pdb]
STPC$S	STPC$S [pdb]
STPC$P	STPC$P [pdb]

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
```

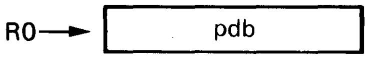
pdb

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 or #0
```

Argument Block

The calling argument block generated (or assumed to exist) by the STPC$x macro has the following format:



MLO–475–87

Syntax Example

```
STPC$ area=#STPARG,pdb=#NGPROC
```

Semantics

If the subject process does not have a stop flag in effect or the subject process is the caller, the STPC$ primitive modifies the subject process's context so that it will begin execution at its termination entry point when it is subsequently scheduled for execution or when control is returned in the case of a "self-stop" request. The primitive also sets an aborted status indication (state-modifier bit SM.ABO or SM.ABP in field PC.STS), which inhibits any later suspension of the subject process. If the subject process is in ready-active state, the primitive returns control to the calling process. If the subject process is the caller, the primitive also returns control to the calling process. If the subject process is not in the ready-active or run state, one of the following cases applies:

- If the subject process is blocked on a semaphore (wait-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.
- If the subject process is blocked on a ring buffer (wait-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.
- If the subject process is in exception-wait-active 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.
- 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.

If the subject process does have a stop flag in effect and is not the caller, the STPC$ primitive sets the process's stop-flag byte to TRUE and returns to the caller.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST	Invalid structure descriptor (index or name); no such process exists. (This error return could be caused by an invalid PDB address.)

3.60 TRLN$ (Translate Logical Name)

Pascal equivalent: TRANSLATE_LOGICAL_NAME Procedure

The Translate Logical Name (TRLN$) primitive allows the caller to obtain the translation value defined for a given logical name. More precisely, the TRLN$ primitive returns the translation string contained in the kernel data structure identified in the call to a specified user-buffer area. Unlike most primitives that operate on existing kernel structures, the TRLN$ primitive performs only one level of translation in the case of "nested" logical-name definitions; the immediate translation value is always returned.

The caller supplies the logical name in a structure descriptor block (SDB) and specifies a buffer area that is to receive the translation-string value. The caller also specifies a maximum length for the returned string. On return from the primitive, the user's length parameter is modified to reflect the actual length of the returned string. (A translation string can be up to 256 bytes in length.)

The complementary Create Logical Name (CRLN$) primitive defines the translation value associated with a logical name, and the Delete Logical Name (DLLN$) primitive eliminates the translation value associated with a currently defined logical name.

Syntax

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

Variant	Macro Call
TRLN$	TRLN$ [area,sdb,string,lenptr]
TRLN$S	TRLN$S [sdb,string,lenptr]
TRLN$P	TRLN$P [sdb,string,lenptr]

area

The address of a user-memory location at 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 user-constructed structure descriptor block (SDB) containing the logical name to be translated. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

```
[SDB=]sdb-address
```

string

The address of a user-memory area that is to receive the ASCII character string defined as the translation value for the logical name. (The effective maximum size of the area is determined by the lenptr parameter.) This argument has the form:

```
[STRING=]area-address
```

lenptr

The address of a word that specifies the maximum length in bytes of the character string to be returned at the location pointed to by the string argument. The valid range of the length value is 1 to 256(decimal). (Location lenptr is updated to reflect the actual string length.) This argument has the form:

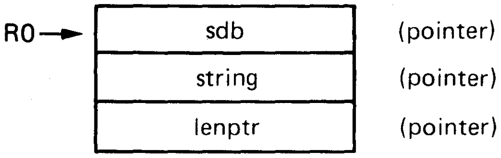
```
[LENPTR=]word-address
```

Restrictions

The locations specified by both the string and lenptr arguments must be in read/write memory.

Argument Block

The calling argument block generated (or assumed to exist) by the TRLN$x macro has the following format:



R0 →	sdb	(pointer)
	string	(pointer)
	lenptr	(pointer)

MLO–476–87

Syntax Example

```
TRLN$S sdb=#LGNAME,string=#TRANS,lenptr=#MAXACT
```

Semantics

The TRLN$ primitive verifies that the kernel data structure identified by the passed SDB is of type ST.LNM (logical name) and tests the maximum length pointed to in the call for a value at least equal to the length of the translation string. If no error is encountered, the primitive copies the translation string contained in the kernel structure to the caller's buffer, places the actual string length in location lenptr, and returns to the caller.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; string-buffer, length-pointer, or SDB address is not on a word boundary or is not in the user's address space. (The addresses are checked only if the CHECK option is selected in the configuration file.)

ES$IPM Illegal parameter; the specified maximum string length is less than the actual string length.

ES$IST Invalid structure description (index or name); no such logical name exists. (This error return could be caused by an erroneous SDB address if address checking is not in effect.)

Implementation Notes

Since logical names are, by nature, often dynamically redefined and each redefinition can cause a new logical-name structure to be created, the structure index contained in a "used" SDB for a logical name could become obsolete between successive uses of the SDB. As a precautionary measure, therefore, the structure index (first word) of the SDB should be cleared before each translation call to prevent possible invalid results. (As explained in Section 3.1.5, the presence of a nonzero structure index causes the reference by structure name to be bypassed in favor of the faster reference by structure ID.)

3.61 UMAP$ (Unmap Window)

Pascal equivalent: UNMAP_WINDOW Procedure

The Unmap Window (UMAP$) primitive permits a process to reverse the effect of a prior MAPW$ operation, disassociating a sequence of virtual addresses (the virtual window) from the physical memory to which it was mapped. (The primitive is valid only in a mapped environment.) More precisely, UMAP$ sets the APR(s) corresponding to a specified window to inactive, or "no access," and modifies the calling process's mapping context to reflect the availability of the APR(s) for subsequent remapping.

The caller identifies the window to be unmapped by supplying the base virtual address of the window and a length to unmap. The address is presumably one previously returned by the MAPW$ primitive. The MAPW$ primitive provides the complementary window-mapping operation. An explicit unmapping operation is required between successive mapping operations that remap a given window in "free" mode. If the "fixed" mode of MAPW$ operation is used for the remapping, however, intervening UMAP$ calls are unnecessary.

Chapter 5 contains a general discussion of dynamic mapping, including the use of UMAP$ in the context of the related primitives MAPW$, GMAP$, SCTX$, and RCTX$. The SCTX$ and RCTX$ primitives provide additional support for mapping operations that involve "borrowing" of one or more APRs.

Syntax

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

Variant	Macro Call
UMAP$	UMAP$ [area,wptr,len,opt]
UMAP$S	UMAP$S [wptr,len,opt]
UMAP$P	UMAP$P [wptr,len,opt]

area

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

```
[AREA=]arg-blk-address
```

wptr

The address of a word in user memory (the window-pointer location) containing the virtual address that identifies the window to be unmapped. Normally, the value in location wptr is the value supplied by a prior MAPW$ call. This argument has the form:

```
[WPTR=]word-address
```

len

An unsigned integer representing the length in bytes of the virtual window to be unmapped. The len value effectively determines the number of APRs that are unmapped, or freed, by the operation. This argument has the form:

```
[LEN=]integer
```

opt

A predefined bit-mask symbol specifying an optional feature of the unmapping operation for a target environment that supports I&D-space separation, such as an LSI–11/73. (The symbol value or default produces a bit-mask word in the calling argument block.) The alternative option symbols, defined by the RIBDF$ macro, and their meaning are:

WD$INS—The operation modifies the process's instruction-space APR set **or**
WD$DAT—The operation modifies the process's data-space APR set; the default.

The option argument is meaningful only if I&D-space separation, possible in an LSI–11/73 or similar target system, is in effect for the calling process. Otherwise, the argument value is ignored. This argument has the form:

```
[OPT=]option-symbol
```

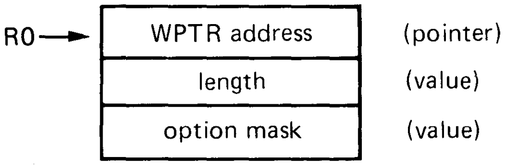
The argument may be null, implying the WD$DAT option default.

Restrictions

This primitive may be used only at process level; that is, it may not be called from an ISR fork routine.

Argument Block

The calling argument block generated (or assumed to exist) by the UMAP$x macro has the following format:



MLO–477–87

Syntax Example

```
UMAP$S wptr=#WINDOW,len=#30000
```

This stack ($S) form of the macro call requests the unmapping of a 30000(octal)-byte (12KB) window whose initial virtual address, or window pointer, is contained in location WINDOW. The length value, corresponding to one and a half virtual pages, implies that two consecutive APRs will be freed for subsequent reuse. (See the MAPW$ primitive description for the corresponding window-mapping example.)

Semantics

The UMAP$ primitive determines which APR(s) map the window identified in the request and clears the corresponding PDR(s), effectively setting the access control field of the affected APR(s) to "no access." The unmapping operation is performed on both the MMU hardware registers and the corresponding locations in the mapping-context restore area associated with the caller's PCB.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IPR	Illegal primitive call; the primitive service was requested in an unmapped environment.

3.62 WAIA$ (Wait on Any Semaphore)

Pascal equivalent: WAIT_ANY Procedure

The Wait on Any Semaphore (WAIA$) primitive implements a complex form of the Wait on Semaphore operation; see the WAIT$ and SGNL$ primitives for a description of the basic Wait and Signal operations. WAIA$ performs the basic Wait operation on the logical OR of several binary or counting semaphores, with an optional timeout feature. That is, WAIA$ permits the calling process to test for and, if necessary, wait on a signal on any one of a set of binary or counting semaphores. Up to four such semaphores may be specified in the primitive request. If none of the specified semaphores can be immediately decremented, the calling process blocks until any one of those semaphores is signaled and can be decremented on behalf of the calling process. (The caller could be blocked behind other waiting processes on a given semaphore, of course, although a multiple-waiter policy is unlikely, particularly in the case of complex-primitive usage.)

Optionally, the Wait Any operation can be terminated because of the expiration of a time interval specified in the request. On successful return from the operation (C bit clear), R0 will contain either an ordinal value identifying the semaphore that satisfied the request or a 0, indicating that the request timed out.

Thus, the WAIA$ primitive allows a process to synchronize with any of up to four events, each signaled by a separate process, for example. The primitive might also be used primarily for its timeout capability, regardless of the number of semaphores involved.

If a zero time period (immediate timeout) is specified in the request, the WAIA$ primitive provides a complex form of the Conditional Wait on Semaphore (WAIC$) operation, which tests for a signaled semaphore but will not block the caller. See the WAIC$ primitive for a description of the simple Conditional Wait operation.

Syntax

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

Variant	Macro Call
WAIA$	WAIA$ [area,time,sdb1,sdb2,sdb3,sdb4]
WAIA$S	WAIA$S [time,sdb1,sdb2,sdb3,sdb4]
WAIA$P	WAIA$P [time,sdb1,sdb2,sdb3,sdb4]

area

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

```
[AREA=]arg-blk-address
```

time

The address of a 2-word user-memory location that specifies a timeout interval, expressed in milliseconds. The first word of the double-precision integer contains the low-order portion of the time value, and the second word (time+2) contains the high-order portion.

An argument value of 0 implies no timeout for the request; the calling process may block indefinitely. This argument has the form:

```
[TIME=]word-address or #0
```

If the address value is nonzero but the time value pointed to is zero, the request will be timed out immediately if none of the specified semaphores can be decremented without waiting. That is, the calling process will never block if the specified time interval is 0.

sdb-i

The address of a structure descriptor block (SDB) that identifies one of the semaphores to be operated on. From one to four SDB addresses may be specified. The order in which the SDBs are specified (or are identified if enumerated by keyword) determines the order in which the corresponding semaphores are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Implementation Notes below.) The sdb-i arguments have the form:

```
[SDBi=]sdb-address
```

Where i may have the value 1 through 4 if the keyword form of argument is used.

Restrictions

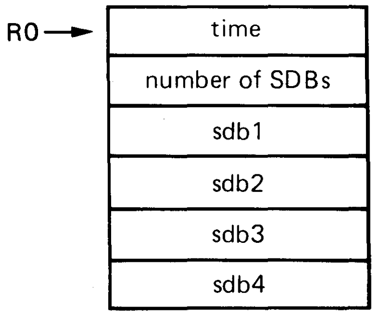
The semaphore identified by a passed SDB must not be a queue semaphore. (Binary and counting semaphores may be "intermixed" in the same request.)

The timeout value may not exceed (2**31)–1, the largest positive integer expressible in 32 bits. That is, the sign bit of the time-interval doubleword (bit 15 of the high-order word) must not be set. (The maximum valid value, in milliseconds, permits a timeout period of just over 24.89 days; see the SLEP$ primitive for more detail.)

If the keyword form of macro call is used, higher-numbered sdb-i keywords may not be used unless each of the lower-numbered sdb-i keywords is specified. That is, if the keyword sequence contains SDB3=, for example, the sequence must also include SDB1= and SDB2=, though not necessarily in numeric order.

Argument Block

The calling argument block generated (or assumed to exist) by the WAIA$x macro has the following format:



The number of SDB-address fields is variable and is indicated by the value in the second word of the block

MLO–478–87

Syntax Example

```
WAIA$ area=#WAARGS,time=#WAKEUP,sdb1=#HILIMT,sdb2=#LOLMIT
```

Semantics

The WAIA$ primitive tests each semaphore specified in the request for a gate-variable value greater than 0. That is, the primitive tests for a semaphore that is "open" and that can be decremented. (The semaphores are tested in the order in which they are identified in the call, either by position or by keyword value.) If any of the semaphores are open at the time of the call, the primitive decrements the first open semaphore encountered and returns immediately to the caller, with a nonzero value in R0. The R0 value, an integer between 1 and 4, indicates that the nth semaphore identified in the call was decremented.

If none of the semaphores is open and either no timer value or a nonzero timer value was supplied in the call, the primitive switches the calling process to the wait-active state. In that state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores is open and a zero timer value was supplied in the call, the primitive returns immediately to the caller, with a zero value in R0, indicating a timeout. (The calling process thus never leaves the run state in the case of an immediate timeout.)

If the calling process switches to wait-active state, the process is blocked from execution until it can be reactivated either by a signal on one of the blocking semaphores (see SGNL$ semantics) or by elapse of the specified timeout period, if any, before a signal occurs. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or the run state, depending on relative process priorities. If unblocked because of a signal, R0 contains the ordinal value (from 1 to 4) of the semaphore that triggered the return, as described above. If unblocked because of a timeout, R0 contains a 0. In either case, the user's C bit is cleared, distinguishing the value returned in R0 from an error-return indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; timer-value pointer is an odd address or is not in the caller's address space.

ES$IST Invalid structure description (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

ES$IPM Illegal parameter; timer value out of range.

Implementation Notes

Since the initial test of the semaphores for a signal is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, if the relative frequency of signals is high for one of several binary or counting semaphores and the "fast" semaphore is identified as being first, either by position in the SDB sequence or by the keyword SDB1=, signals on that semaphore will tend to mask off the others in a sequence of WAIA$ operations. Thus, the "slower" semaphores may seldom or never be tested and acted on. Optimally, then, the semaphore with the highest expected signal rate should be identified as last, the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so that at least n semaphores are guaranteed to be tested in n calls to WAIA$. The correct or best strategy is application specific.

3.63 WAIC$ (Conditionally Wait on Semaphore)

Pascal equivalent: COND_WAIT Function

The Conditional Wait on Semaphore (WAIC$) primitive decrements the value of a specified binary or counting semaphore if its current value is nonzero, indicating a previous signal, and returns to the caller, with the kernel-defined value TRUE in R0. If the semaphore has not been signaled (current value is already 0) the primitive returns immediately to the caller, with the kernel-defined value FALSE in R0. In no case is the calling process blocked, that is, made to wait until the semaphore is signaled. (Compare with WAIT$, 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 WAIC$ macro and their respective macro calls are listed below. The differences are described in Section 3.1.

Variant	Macro Call
WAIC$	WAIC$ [area,sdb]
WAIC$S	WAIC$S [sdb]
WAIC$P	WAIC$P [sdb]

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 decremented. (See Section 3.1.5 for the format and use of an SDB.) This argument has the form:

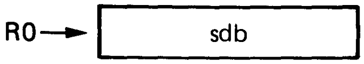
```
[SDB=]sdb-address
```

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 WAIC$x macro has the following format:



MLO-479-87

Syntax Example

```
WAIARG: WAIC$P sdb=BSEM
```

This parameters-only ($P) form of the macro call creates a calling argument block at assembly time, presumably in a pure-data program section, for run-time reference by means of a "WAIC$ AREA=WAIARG" form of call.

Semantics

The WAIC$ primitive decrements the specified semaphore variable if its current value is greater than 0 and returns to the caller, with the value TRUE in R0. If the semaphore value is already 0, the WAIC$ primitive returns immediately to the caller, with the value FALSE in R0. The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

The WAIC$ primitive is a "decrement open (currently signaled) semaphore only" function, returning a successful/unsuccessful indication and never causing the caller to block. Therefore, the primitive could be used, for example, in an ISR fork routine, which is prohibited from blocking for any reason.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST	Invalid structure descriptor (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

3.64 WAIQ$ (Wait on Queue Semaphore)

Pascal equivalent: GET_PACKET Procedure

The Wait on Queue Semaphore (WAIQ$) 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 WAQC$, 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 ALPK$ and SGLQ$ primitives and Section 2.2.2. When no longer needed, a packet obtained through a WAIQ$ request can be returned to the kernel (freed for reuse) by means of the DAPK$ primitive.

In a mapped environment, general and device-access processes do not have direct access to packet content, since they are not mapped to kernel data space. Such processes cannot fetch data from a packet, for example. Therefore, if such a process needs to extract data from an acquired packet rather than simply "pass along" the packet pointer to another process, it must use the RCVD$ primitive. RCVD$ is a higher-level primitive that provides a data-copying and packet-deletion service in addition to the functionality of WAIQ$. (Use of RCVD$ presumes that the packet content is of the form defined by the SEND$ primitive, the higher-level version of SGLQ$.)

The SGLQ$ call is the inverse of the WAIQ$ call.

Syntax

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

Variant	Macro Call
WAIQ$	WAIQ$ [area,sdb,qelm]
WAIQ$S	WAIQ$S [sdb,qelm]
WAIQ$P	WAIQ$P [sdb]

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 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*($S) version of the macro call, the returned pointer value is left on the stack. In the parameters-only ($P) 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.)

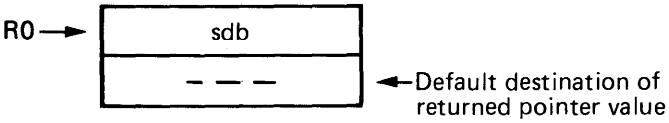
Restrictions

The argument block must be in read/write memory.

You can use the parameters-only ($P) 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 $P call in the RAM portion of a ROM/RAM system unless the argument values are filled in at run time.

Argument Block

The calling argument block generated (or assumed to exist) by the WAIQ$x macro has the following format:



MLO–480–87

Syntax Example

```
WAIQ$ area=#WARGS,qelm=#PKTPTR
```

Semantics

The WAIQ$ primitive tests the specified queue semaphore for an available packet. If at least one packet is on the semaphore's packet queue, the primitive decrements the semaphore's counter value, 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 on the semaphore and calls the scheduler. The process remains on the semaphore's waiting-process list until unblocked by a signal of the semaphore, which places a packet on the queue.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address.)

Applications

See the SGLQ$ description.

3.65 WAIT$ (Wait on Semaphore)

Pascal equivalent: WAIT Procedure

The Wait on Semaphore (WAIT$) primitive attempts to decrement the value of a specified binary or counting semaphore. If successful (the value was not already 0), the calling process proceeds. If unsuccessful (the value was already 0), the calling process is blocked; that is, made to wait until the semaphore is signaled and can be decremented.

This primitive permits the calling process to receive a signal from another process that an event it is dependent on has occurred. Or, viewing a binary semaphore as a gate that may be open or shut, the Wait operation permits the calling process to "close the gate" behind itself with a WAIT$ call before entering a critical section of code, subsequently opening the gate with a SGNL$ call following execution of the critical section. A related process that at some point depends on the former process not being in its critical section can wait on the same semaphore and then signal the semaphore on completion of its own critical section.

Together, the WAIT$ and SGNL$ calls allow two or more cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms (see also WAIC$, SGLC$, and SALL$).

Syntax

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

Variant	Macro Call
WAIT$	WAIT$ [area,sdb]
WAIT$S	WAIT$S [sdb]
WAIT$P	WAIT$P [sdb]

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 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 WAIT$x macro has the following format:

MLO-481-87

Syntax Example

```
WAIT$S sdb=#CSEM
```

Semantics

The Wait on Semaphore primitive decrements the semaphore variable if its current value is greater than 0 and returns immediately to the caller. If the semaphore value is already 0, the WAIT$ primitive switches the calling process to the wait-active state. In that state, the process is blocked from execution until it can be reactivated by a subsequent signal (see SGNL$ semantics).

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such binary or counting semaphore exists. (This error return could be caused by an invalid SDB address.)

3.66 WAQA$ (Wait on Any Queue Semaphore)

Pascal equivalent: GET_PACKET_ANY Procedure

The Wait on Any Queue Semaphore (WAQA$) primitive implements a complex form of the Wait on Queue Semaphore operation; see the WAIQ$ and SGLQ$ primitives for a description of the basic Wait and Signal operations on queue semaphores. WAQA$ performs the basic Wait, or "get packet," operation on the logical OR of several queue semaphores, with an optional timeout feature. That is, WAQA$ permits the calling process to test for and, if necessary, wait on an available packet on any one of a set of queue semaphores. Up to four queue semaphores may be specified in the primitive request. If no packet is available on any of the specified semaphores, the calling process blocks until any one of those semaphores is signaled and can provide a packet pointer for the calling process. (The caller could be blocked behind other waiting processes on a given queue semaphore, of course, although a multiple-waiter policy is unlikely, particularly in the case of complex-primitive usage.)

Optionally, the get-any-packet operation can be terminated because of the expiration of a time interval specified in the request. On successful return from the operation (C bit clear), R0 will contain either an ordinal value identifying the semaphore that satisfied the request or a 0, indicating that the request timed out.

Thus, the WAQA$ primitive allows a process to get a packet pointer from any of up to four queue semaphores, each signaled by a different process, perhaps. The primitive might also be used primarily for its timeout capability, regardless of the number of packet queues involved.

If a zero time period (immediate timeout) is specified in the request, the WAQA$ primitive provides a complex form of the Conditional Wait on Queue Semaphore (WAQC$) operation, which tests for an available packet but will not block the caller. See the WAQC$ primitive for a description of the simple conditional-get-packet operation.

Syntax

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

Variant	Macro Call
WAQA$	WAQA$ [area,time,qelm,sdb1,sdb2,sdb3,sdb4]
WAQA$S	WAQA$S [time,qelm,sdb1,sdb2,sdb3,sdb4]
WAQA$P	WAQA$P [time,qelm,sdb1,sdb2,sdb3,sdb4]

area

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

```
[AREA=]arg-blk-address
```

time

The address of a 2-word user-memory location that specifies a timeout interval, expressed in milliseconds. The first word of the double-precision integer contains the low-order portion of the time value, and the second word (time+2) contains the high-order portion.

An argument value of 0 implies no timeout for the request; the calling process may block indefinitely. This argument has the form:

```
[TIME=]word-address or #0
```

If the address value is nonzero but the time value pointed to is 0, the request will be timed out immediately if no packet is available on any of the specified semaphores when the primitive is called. That is, the calling process will never block if the specified time interval is 0.

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 argument is null, the packet pointer returned by the primitive is available only in the second word of the argument block. If the argument is null in the stack ($S) version of the macro call, the returned pointer value is left on the top of the stack (@SP).

sdb-i

The address of a structure descriptor block (SDB) that identifies one of the semaphores to be operated on. From one to four SDB addresses may be specified. The order in which the SDBs are specified (or are identified if enumerated by keyword) determines the order in which the corresponding semaphores are initially tested for a signal. (That order can be critical under certain real-time conditions, as discussed under Implementation Notes below.) The sdb-i arguments have the form:

```
[SDBi=]sdb-address
```

The metavariable "i" may have the value 1 through 4 if the keyword form of argument is used.

Restrictions

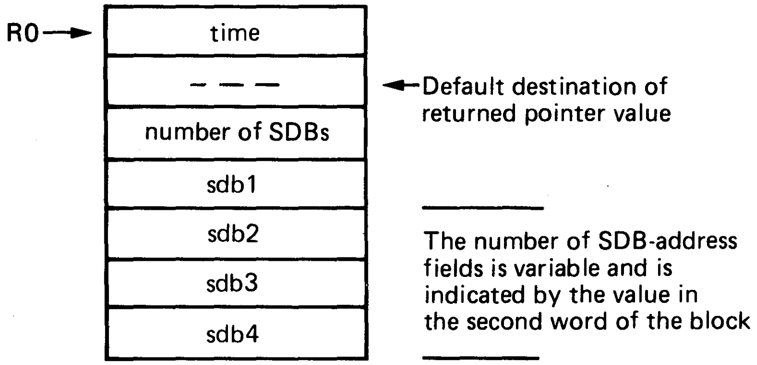
The argument block must be in read/write memory.

The time-out value may not exceed (2**31)–1, the largest positive integer expressible in 32 bits. That is, the sign bit of the time-interval doubleword (bit 15 of the high-order word) must not be set. The maximum valid value, in milliseconds, permits a time-out period of just over 24.89 days (see the SLEP$ primitive for more detail).

If the keyword form of macro call is used, higher-numbered sdb-i keywords may not be used unless each of the lower-numbered sdb-i keywords is specified. That is, if the keyword sequence contains SDB3=, for example, the sequence must also include SDB1= and SDB2=, though not necessarily in numeric order.

Argument Block

The calling argument block generated (or assumed to exist) by the WAQA$x macro has the following format:


R0
time
— — —
Default destination of returned pointer value
number of SDBs
sdb1
sdb2
sdb3
sdb4
The number of SDB-address fields is variable and is indicated by the value in the second word of the block

MLO-482-87

Syntax Example

```
WAQA$S #0,,#UN3RDY,#UN1RDY,#UN2RDY
```

This example shows the stack version of the call with a zero time argument, a null qelm argument, and three SBD addresses. (The request will not be timed out.) The argument block is generated on the stack, and in this case, all fields but qelm are purged on return. The packet pointer returned by the primitive will be available on the top of the stack (@SP) following the call.

Semantics

The WAQA$ primitive tests each semaphore specified in the request for an available queue element, or message packet. (The semaphores are tested in the order in which they are identified in the call, either by position or by keyword value.) If any of the semaphores has a packet at the time of the call, the primitive dequeues a packet pointer from the first such semaphore encountered and returns it immediately to the caller, with a nonzero value in R0. The R0 value, an integer between 1 and 4, indicates that the nth semaphore identified in the call satisfied the request. (The pointer is returned in a manner determined by the form of request used; see Syntax.)

If none of the semaphores has a packet and either no timer value or a nonzero timer value was supplied in the call, the primitive switches the calling process to the wait-active state. In that state, the process is blocked on all the semaphores specified in the request.

If none of the semaphores has a packet and a zero timer value was supplied in the call, the primitive returns immediately to the caller, with a zero value in R0, indicating a timeout. Thus, the calling process never leaves the run state in the case of an immediate timeout.

If the calling process switches to wait-active state, it is blocked from execution until it can be reactivated either by a packet becoming available on one of the blocking semaphores (see SGLQ$ semantics) or by elapse of the specified time-out period, if any. When reactivated for either reason, the process is unblocked from all the semaphores and is switched to either the ready-active or the run state, depending on relative process priorities. If unblocked because of an available packet, R0 contains the ordinal value (from 1 to 4) of the semaphore that triggered the return, as previously described. If unblocked because of a timeout, R0 contains a 0. In either case, the user's C bit is cleared, distinguishing the value returned in R0 from an error-return indication.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception codes that may be returned by the primitive are:

ES$IAD Invalid address; timer-value pointer is an odd address.

ES$IST Invalid structure description (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address.)

ES$IPM Illegal parameter; timer value out of range.

Implementation Notes

Since the initial test of the semaphores for an available packet is performed in determinate order, the order in which multiple semaphores are identified in the call can be critical under certain real-time conditions. For example, if the relative frequency of signals or sends is high for one of several queue semaphores and the "fast" semaphore is identified as being first, either by position in the SDB sequence or by the keyword SDB1=, that semaphore will tend to mask off the others in a sequence of WAQA$ operations. Thus, the "slower" semaphores may seldom or never be tested and serviced. Optimally, then, the semaphore with the highest expected signal rate should be identified as last, the next highest as next to last, and so on, assuming that probable relative frequencies can be determined. Alternatively, the order in which the semaphores are identified could be rotated in successive calls so that at least n semaphores are guaranteed to be tested in n calls to WAQA$. The correct or best strategy is application specific.

3.67 WAQC$ (Conditional Wait on Queue Semaphore)

Pascal equivalent: COND_GET_PACKET Function

The Conditional Wait on Queue Semaphore (WAQC$) primitive tests the specified semaphore for an available packet. If one is available, the primitive removes the packet from the semaphore's packet queue, returns the packet pointer to the caller, and returns the kernel-defined value TRUE (1) in R0. If no packet is available, the primitive returns immediately to the caller, with the kernel-defined value FALSE (0) 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 has not already occurred. (Compare with WAIQ$, the unconditional Wait on Queue call.)

See the SGLQ$ primitive and Section 2.2.2 for a description of queue packets. When no longer needed, a packet obtained through a WAQC$ request can be returned to the kernel (freed for reuse) by means of the DAPK$ primitive.

In a mapped environment, general and device-access processes do not have direct access to packet content, since they are not mapped to kernel data space. Such processes cannot fetch data from a packet, for example. Therefore, if such a process needs to extract data from an acquired packet, it must use the corresponding RCVC$ primitive. RCVC$ is a higher-level primitive that provides a data-copying and packet-deletion service in addition to the functionality of WAQC$. (Use of RCVC$ presumes that the packet content is of the form defined by the SEND$ primitive, the higher-level version of SGLQ$.)

The inverse of the WAQC$ call is the SGQC$ call.

Syntax

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

Variant	Macro Call
WAQC$	WAQC$ [area,sdb,qelm]
WAQC$S	WAQC$S [sdb,qelm]
WAQC$P	WAQC$P [sdb]

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.

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 ($S) version of the macro call, the returned pointer value is left on the stack. In the parameters-only ($P) 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.)

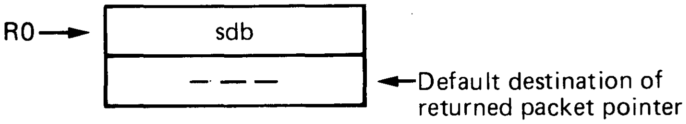
Restrictions

The argument block must be in read/write memory.

You can use the parameters-only ($P) 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 $P call in the RAM portion of a ROM/RAM system unless the argument values are filled in at run time.

Argument Block

The calling argument block generated (or assumed to exist) by the WAQC$x macro has the following format:



MLO–483–87

Syntax Example

```
WAQC$S 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 dequeues the first available packet, decrements the semaphore's counter value, and returns the address of that packet in the last word of the argument block. If requested (qelm argument), the macro expansion moves the packet pointer from the argument block to a user-specified location. The primitive also returns the value TRUE in R0, signifying that a packet was obtained.

If no packets are on the semaphore queue, the primitive returns to the caller, with the value FALSE in R0, signifying that no packet was obtained. The TRUE and FALSE symbol values are defined by the EXMSK$ macro in the COMM and COMU libraries. Those values are 1 and 0, respectively, in the current version of MicroPower/Pascal.

Error Returns

See Section 3.1.4 for general information about error returns. The specific exception code that may be returned by the primitive is:

ES$IST Invalid structure descriptor (index or name); no such queue semaphore exists. (This error return could be caused by an invalid SDB address.)

Chapter 4
System Configuration Macros

The first step in building a new application image for a given target system is to create a system configuration file. That file must contain the configuration macro calls needed to describe the target system hardware and to specify the kernel software parameters desired for your specific application. (Often you can create a suitable configuration file by modifying one of the configuration files included in the distributed MicroPower/Pascal software.) In the kernel-build phase of application building, you assemble the configuration file with the COMM or COMU macro library and merge the resulting object module with the PAXM or PAXU kernel object library to produce a "tailored" kernel object module. The entire process of application building is described in the MicroPower/Pascal system user's guide for your host system.

Table 4–1 summarizes the basic functions of the configuration macros from the user's viewpoint.

Table 4–1: Configuration Macro Functions

Macro	Function
CONFIGURATION	Identifies the file as a system configuration file; mandatory, must be the first macro invoked in the file.
SYSTEM	Determines whether the kernel should be optimized as specified by optional RESOURCES, PRIMITIVES, and TRAPS macro calls and whether the kernel should include the debugger service module; mandatory, must be invoked as the second macro in the file.
PROCESSOR	Describes the type of target processor and some of its hardware characteristics; mandatory, must be invoked as the third macro in the file.
MEMORY	Describes the location, size, and characteristics of a uniform segment of target memory; mandatory, may be invoked more than once.
DEVICES	Specifies the set of I/O interrupt vectors used in the target system (six vectors in each call); mandatory, may be invoked more than once.
FALCON	Describes hardware characteristics and the trap-handling options that are specific to an SBC–11/21 (FALCON or FALCON–PLUS) target.

Table 4-1 (Cont.): Configuration Macro Functions

Macro	Function
KXT11C	Describes hardware characteristics and the trap-handling options that are specific to a KXT11-CA target.
KXJ11C	Describes hardware characteristics and the trap-handling options that are specific to a KXJ11-CA target.
RESOURCES	Specifies the amount of RAM memory to be allocated to the kernel for its stack, for message packets, and for dynamic data structures; may be defaulted.
PRIMITIVES	Specifies the set of primitive service modules to be included in the kernel; may be defaulted; may be invoked more than once.
TRAPS	Specifies trap processors to be included in the kernel (eight traps in each call); may be defaulted; may be invoked more than once.
LOGICAL	Declares a logical name and its translation string; optional; may be invoked more than once.
ENDCFG	Ends the system configuration file; mandatory, must be the last macro in the file.

The remainder of this chapter discusses the general functions of a configuration file, describes several prototype configuration files, and describes the individual configuration macros in detail.

4.1 Functions of the Configuration File

A system configuration file has several interlocking functions:

- Provides hardware configuration information about the target system for use by the kernel and the build utility programs
- Supplies the application-specific information required for configuring the kernel
- Generates the global symbol references that "pull" all the kernel modules needed by a particular application from the kernel object module library (PAXM or PAXU)

The hardware and kernel-software configuration information includes the following:

- The type of target processor and its hardware options—described by the PROCESSOR macro and, possibly, the FALCON, KXT11C, or KXJ11C macro
- The target memory configuration—described by MEMORY macros
- The interrupt vector locations used by all devices installed on the target system—specified by DEVICES macros
- The need for a debugger service module to be included in the kernel image (used by PASDBG)—specified by the SYSTEM macro
- The trap-processing modules to be included in the kernel—implied or specified by any of the SYSTEM, PROCESSOR, MEMORY, FALCON, KXT11C, KXJ11C, and TRAPS macros

- The amount of read/write memory to be allocated in the kernel's impure area for its stack, for message packets, and for dynamic data structures—implied by the SYSTEM macro or specified by the RESOURCES macro
- The primitive service modules to be included in the kernel—implied by the SYSTEM macro or specified by the PRIMITIVES macro

Six configuration macros must be included in any configuration file; the other macros are used as needed. The first three macros in the file must appear in the following order: CONFIGURATION, SYSTEM, and PROCESSOR; and the last macro must be ENDCFG. You must also include the MEMORY and DEVICES macros. In summary, the configuration file must contain at least the following macros:

CONFIGURATION
SYSTEM
PROCESSOR
MEMORY
DEVICES
ENDCFG

In addition, if the processor type is FALC or FALCPLUS, you must include the FALCON macro in the file. Similarly, if the processor type is KXT11C or KXJ11C, you must include, respectively, the KXT11C or KXJ11C macro in the file. You may or may not need the RESOURCES, PRIMITIVES, and TRAPS macros, depending on an option specified in the SYSTEM macro. You only need one or more LOGICAL macros if you are defining logical names at build time.

4.2 Prototype Configuration Files CFDxxx.MAC

The MicroPower/Pascal distribution kit includes configuration files that are used for software-installation verification. Such files have names of the form CFDxxx.MAC and are as follows:

- CFDCMR.MAC for a CMR21 target with 32KB of RAM
- CFDFAL.MAC for a FALCON target with 32KB of RAM and the KXT11–A2 firmware
- CFDFPL.MAC for a FALCON–PLUS target with 32KB of RAM and the KXT11–A5 firmware
- CFDKJU.MAC for an unmapped KXJ11–CA target with 56KB of RAM
- CFDKTC.MAC for a KXT11–CA target with 32KB of native RAM (map 0)
- CFDMAP.MAC for a mapped LSI–11/23 target with 64KB of RAM
- CFDUNM.MAC for any unmapped LSI–11 target with 32KB of RAM

Each of the files assumes a hardware configuration that will allow the minimal CARS3 application image example to be loaded under PASDBG and, wherever possible, assumes the factory-standard settings for configurable options. Any of the files can be used as a prototype, or editing base, for development of a configuration file for a real application. A copy of CFDFPL.MAC might be modified, for example, to develop a "tailored" configuration file for a FALCON–PLUS application. After you familiarize yourself with each of the configuration macros that applies to your type of target system, inspect the relevant CFDxxx.MAC file to see how it needs to be modified for your target and application.

Note

In MACRO–11 assembly language notation, a number in the source code representing a decimal integer value must be terminated by a decimal point. MACRO–11 interprets numbers not ending with a decimal point as octal values. Also, angle brackets are used instead of parentheses to delimit an expression.

4.3 Configuration Macro Calls

The configuration macro calls are defined in alphabetical order in the following sections.

Note

In the descriptive text, all numeric values except addresses are expressed in decimal unless otherwise indicated. Addresses are expressed in octal. In the macro syntax definitions and examples, however, MACRO–11 notation rules apply: decimal numbers end with a decimal point; octal numbers do not.

4.3.1 CONFIGURATION Macro

The CONFIGURATION macro initializes a configuration file. (A .MCALL CONFIGURATION directive is needed preceding the CONFIGURATION call, but no .MCALL directives are required for the other configuration macros.) The macro takes one optional argument, name, which generates a .IDENT directive for the configuration file. (The .IDENT directive is an assembler directive that produces a version-identifier record in the resulting object module.)

The CONFIGURATION macro must be the first macro invoked in a configuration file.

Syntax

CONFIGURATION [name]

Arguments

name

An identifier of up to six characters, each of which must be a valid RAD50 character (uppercase alphabetics and numerics only). If you do not specify the argument, no .IDENT statement is generated, and the MERGE utility defaults the version identifier of the merged object module to the first such identifier it finds in the modules being merged from the kernel library.

Example

```
CONFIGURATION  KRNV05
```

4.3.2 DEVICES Macro

The DEVICES macro defines the set of I/O interrupt vectors used by devices installed on the target system. You can specify up to six vectors in each statement.

Interrupt vectors are locations in low memory that contain the address of a device's interrupt dispatch block (IDB) and the new processor status word. Each device must be installed so that it interrupts at one of the specified vector addresses.

The DEVICES macro allocates a unique IDB for each vector specified in the macro. Since a vector may be in ROM, the vector must permanently point to its IDB, through which the kernel dispatches interrupts to interrupt service routines (ISRs). In order to permit run-time connection of ISRs to interrupts, IDBs therefore must be allocated in RAM, and such allocation must be done during the build procedure. See Chapter 7 for a full description of IDBs and interrupt dispatching. If a vector is specified in DEVICES but is not used by the application, some memory space is wasted because of the unneeded IDB.

Note

> When building an all-RAM application image with debugging support, you do not need to specify the console-terminal vector addresses, 60 and 64, which are used by the debugger's host-to-target serial line. The DSM and PASDBG will preempt those vectors when the application begins execution, whether or not they are specified in DEVICES.

Syntax

DEVICES v1,v2,...v6

Arguments

vn

The address of the first word of each vector used by the application. You can specify up to six addresses in each DEVICES macro and can use as many DEVICES calls as needed. The minimum allowable vector address is 0; the maximum allowable vector address is 1000 for an LSI-family or J11-based processor (other than a KXJ11-CA) and 400 for an SBC-11/21, KXT11-CA, KXJ11-CA, or CMR21 processor. See also the discussion of the PROCESSOR macro.

Example

```
DEVICES  60,64,100,300,310,320
```

4.3.3 ENDCFG Macro

The ENDCFG macro terminates a system configuration file. (The macro performs various error checks and other implicit functions.) ENDCFG must be the last macro invoked in a configuration file.

Syntax

ENDCFG

4.3.4 FALCON Macro

The FALCON macro must be used in a configuration file for an SBC–11/21 FALCON or FALCON–PLUS target if the FALC or FALCPLUS processor type is specified in the PROCESSOR macro. The FALCON macro has the following functions:

- Describes how interrupt vector 140 is configured on the processor board—whether an SLU1 break (bus BHALT) signal or a nonexistent-memory/timeout (NXM) error causes a level-7 trap through 140. The board can be jumpered to select either action for a MicroPower/Pascal application; the break trap to vector 140 is the standard factory configuration, with the NXM condition trapping to the restart address.
- Selects the action to be taken by the kernel for an SLU1 break and, by implication, for a HALT instruction. The range of possibilities for break handling depends on the vector 140 configuration.

For a MicroPower/Pascal application, the hardware configuration for break signals and NXM errors is assumed to be either the factory standard or its exact opposite.

For either configuration, the possible options for break handling include a transfer to ROM ODT or a transfer to software ODT, with HALTs also transferring to ODT. If a break traps to 140, the additional possibilities for break handling are an ES$BRK exception or a "no action" response—an immediate return from the trap (IGNORE)—with HALTs causing a processor "hang" (BR . instruction loop). If an NXM error traps to 140, the additional possibility for break handling is a processor hang, with HALTs also causing a hang.

In any case, an NXM error results in a simulated ES$BUS (trap to 4) exception, of type EX$MEM.

Syntax

$$\text{FALCON trap140=} \left\{ \begin{array}{l} \text{BHALT} \\ \text{NXM} \end{array} \right\} \text{, break=} \left\{ \begin{array}{l} \text{ROMODT} \\ \text{SFWODT} \\ \text{EXCEPTION} \\ \text{IGNORE} \\ \text{HANG} \end{array} \right\}$$

Arguments

trap140=

TRAP140=BHALT specifies that a BHALT bus signal, whether generated by the BREAK key on a terminal connected to SLU1 or asserted by an LSI–11 bus (Q-bus) device, will cause a level-7 trap through vector 140 and that NXM will trap to the restart address.

TRAP140=NXM specifies that a nonexistent-memory/timeout error will cause a level-7 trap through vector 140 and that a break will trap to the restart address.

The default is TRAP140=BHALT.

break=

BREAK=ROMODT selects a transfer to ROM ODT as the action to be taken by the hardware or the kernel for a break condition, implying that a Macro-ODT ROM chip set (KXT11–A2 or KXT11–A5 option) is properly installed on the target processor. Note that this option also implies that a HALT instruction causes a transfer to ROM ODT.

BREAK=SFWODT selects a transfer to software ODT as the action to be taken by the kernel for a break condition and causes the optional software module FALODT to be included in the kernel. The FALODT module implements an ODT command set that is very similar to Macro-ODT; see the MicroPower/Pascal installation guide for your host system. The module occupies approximately 666 bytes of ROM and on a FALCON–PLUS target uses 128 bytes of additional kernel RAM allocated by this option. Note that this option also implies that a HALT instruction causes a transfer to software ODT.

Note

> If you select the BREAK=SFWODT option, you must also select the proper baud rate for the console line by editing the configuration file for the FALCON (CFDFAL.MAC) or FALCON–PLUS (CFDFPL.MAC). Instructions are provided in those files.

BREAK=EXCEPTION requests that the kernel raise an ES$BRK exception, of type EX$EXC, for a break trap and implies that a HALT instruction will cause the processor to hang (valid only if trap140=BHALT).

BREAK=IGNORE requests that a break trap be ignored, that is, result in an immediate resumption of normal execution, and implies that a HALT instruction will cause the processor to hang (valid only if trap140=BHALT).

BREAK=HANG requests that a break condition will cause the processor to hang and implies that a HALT instruction will also cause the processor to hang (valid only if trap140=NXM).

The default is BREAK=ROMODT.

Restrictions

Because MicroPower/Pascal requires that some ROM in lowest memory in any ROM/RAM target system must be other than a CMR21, the FALCON–PLUS memory map 3 configuration is not supported.

The break=SFWODT option is applicable only to a ROM/RAM target when it is desirable to retain some console-debugging capability in the PROMed kernel.

Hardware Configuration Assumptions

If the target is RAM-only except for the mandatory Macro-ODT/bootstrap ROM chip set, the starting address must be either 172000 or 173000 for applications to be loaded by PASDBG or booted stand-alone from a target device. (Start-up at 173000 implies a bus reset and a delay prior to reinitialization on a restart; start-up at 172000 does not.)

If the target is ROM/RAM, the starting address must be 0 for start-up and restart in the kernel.

Example

```
FALCON TRAP140=BHALT, BREAK=EXCEPTION
```

4.3.5 KXJ11C Macro

The KXJ11C macro must be used in a configuration file for a KXJ11–CA peripheral processor or stand-alone target if the KXJ11C processor type is specified in the PROCESSOR macro. The KXJ11C macro has the following functions:

- Specifies the action to be taken by the kernel for a BHALT signal on the Q-bus, primarily to facilitate an arbiter/slave debugging situation
- Specifies the action to be taken by the kernel for a BINIT (bus reset) signal on the Q-bus
- Implies special interrupt dispatching for the multiprotocol SLU2 vector

Syntax

$$\text{KXJ11C bhalt=}\left\{\begin{array}{l}\text{YES}\\ \text{NO}\end{array}\right\}\text{, reset=}\left\{\begin{array}{l}\text{IGNORE}\\ \text{BOOT}\\ \text{RSTBOT}\\ \text{INTRPT}\end{array}\right\}$$

Arguments

bhalt=

BHALT=YES specifies that a BHALT signal on the Q-bus, asserted by the arbiter processor, will cause a trap that is directed by the powerup module to kernel location $KXJDB. Location $KXJDB contains an RTS PC instruction, and you can set a PASDBG breakpoint there while debugging to effect a "debug trap" for Q-bus halts. (If no breakpoint is set, the trap has no observable effect; it is dismissed immediately.)

BHALT=NO specifies that a BHALT signal on the Q-bus will not be enabled to cause a trap and will be ignored by the KXJ11–CA.

The default is BHALT=NO.

reset=

RESET=option indicates the action to be taken in the event of a reset signal on the Q-bus:

- IGNORE—Q-bus resets are to be ignored.
- BOOT—Q-bus resets are to cause a KXJ11–CA powerup.
- RSTBOT—Q-bus resets are to cause a KXJ11–CA reset and then a powerup.
- INTRPT—Q-bus resets are to cause a simulated hardware interrupt through vector 220. You can connect an ISR to the interrupt and do any processing needed by the application. When finished, you can resume application execution by exiting from the interrupt service routine with an RTS PC instruction. Alternatively, you can restart the application by branching to location $KXJPC.

The default is RESET=IGNORE.

The KXJ11C macro also causes a special interrupt dispatching module (KSLU2) to be included in the kernel. The module effectively "splits" the standard multiprotocol-chip SLU2 vector into four separate pseudovectors for each channel. The emulated vectors for SLU2 device interrupts are the following:

- 140—Channel A receive character interrupt
- 144—Channel A transmit character interrupt
- 150—Channel A receive error interrupt
- 154—Channel A modem control interrupt
- 160—Channel B receive character interrupt
- 164—Channel B transmit character interrupt
- 170—Channel B receive error interrupt
- 174—Channel B modem control interrupt

The TT, DD, and XS drivers depend on those pseudovectors and should therefore be specified in the DEVICES macro instead of vector 70 if SLU2 channel A or B is used.

4.3.6 KXT11C Macro

The KXT11C macro must be used in a configuration file for a KXT11-CA peripheral processor or stand-alone target if the KXT11C processor type is specified in the PROCESSOR macro. The KXT11C macro has the following functions:

- Specifies the action to be taken by the kernel for a BHALT signal on the Q-bus, primarily to facilitate an arbiter/slave debugging situation
- Specifies the action to be taken by the kernel for a BINIT (bus reset) signal on the Q-bus
- Describes the target's memory map configuration
- Implies special interrupt dispatching for the multiprotocol SLU2 vector

Syntax

$$\text{KXT11C bhalt=}\left\{\begin{array}{c}\text{YES}\\ \text{NO}\end{array}\right\}\text{, reset=}\left\{\begin{array}{c}\text{IGNORE}\\ \text{BOOT}\\ \text{RSTBOT}\\ \text{INTRPT}\end{array}\right\}\text{, map=n}$$

Arguments

bhalt=

BHALT=YES specifies that a BHALT signal on the Q-bus, asserted by the arbiter processor, will cause a trap that is directed by the powerup module to kernel location $KXTDB. Location $KXTDB contains an RTS PC instruction, and you can set a PASDBG breakpoint there while debugging to effect a "debug trap" for Q-bus halts. (If no breakpoint is set, the trap has no observable effect; it is dismissed immediately.)

BHALT=NO specifies that a BHALT signal on the Q-bus will not be enabled to cause a trap and will be ignored by the KXT11-CA.

The default is BHALT=NO.

reset=

RESET=option indicates the action to be taken in the event of a reset signal on the Q-bus:

- IGNORE—Q-bus resets are to be ignored.
- BOOT—Q-bus resets are to cause a KXT11-CA powerup.
- RSTBOT—Q-bus resets are to cause a KXT11-CA reset and then a powerup.
- INTRPT—Q-bus resets are to cause a simulated hardware interrupt through vector 220. You can connect an ISR to the interrupt and do any processing needed by the application. When finished, you can resume application execution by exiting from the interrupt service routine with an RTS PC instruction. Alternatively, you can restart the application by branching to location $KXTPU.

The default is IGNORE.

map=n

MAP=n describes the KXT11-CA memory map configuration. The map parameter, n, must be an integer between 0 and 7 that corresponds to the memory-map jumper settings on the KXT11-CA. The default is 0. By implication, the parameter indicates the location of the high-order 64 bytes of native RAM that will be used by firmware. (Those last 64 bytes of native RAM must not be described in the MEMORY macro for that RAM segment.)

Note

> The functionality of the POWER argument in earlier versions of the KXT11C macro, and more specifically of the POWER=NONVOL option, has been shifted to the VOLATILE argument of the MEMORY macro in MicroPower/Pascal V2.0 and later versions. If the VOLATILE=NO option is specified in the MEMORY macro for a RAM memory segment, warm-restart capability is included in the kernel, as described under the MEMORY macro. See also the PWFL$ primitive description in Chapter 3 for more discussion of warm restarts, as well as the special rules at the end of Section 4.3.7 about debugging a KXT11-CA target declared as having nonvolatile RAM.

The KXT11C macro also causes a special interrupt dispatching module (KSLU2) to be included in the kernel. The module effectively "splits" the standard multiprotocol-chip SLU2 vector into four separate pseudovectors for each channel. The emulated vectors for SLU2 device interrupts are the following:

- 140—Channel A receive character interrupt
- 144—Channel A transmit character interrupt
- 150—Channel A receive error interrupt
- 154—Channel A modem control interrupt
- 160—Channel B receive character interrupt
- 164—Channel B transmit character interrupt
- 170—Channel B receive error interrupt
- 174—Channel B modem control interrupt

The TT, DD, and XS drivers depend on those pseudovectors and should therefore be specified in the DEVICES macro instead of vector 70 if SLU2 channel A or B is used.

4.3.7 LOGICAL Macro

The LOGICAL macro specifies a build-time logical name and its translation string. At run time, the kernel will create the logical names, if any, specified in the configuration file before any static process is started. Thus, the LOGICAL macro facilitates the use of logical names that are determined at build time in the initialization code of static processes.

The LOGICAL macro may be invoked as many times as needed.

Syntax

LOGICAL name=xxxxxx, string=yyyyyyyyyy

Arguments

name=xxxxxx

A logical name of up to six ASCII characters that is to be associated at run time with the translation value specified by the string argument. (By system convention, a logical name—like all run-time structure names—is normally six characters in length and is padded with trailing space characters if it contains fewer than six printing characters.) Logical names and translation strings are case sensitive.

string=yyyyyyyyyy

The ASCII translation string for the logical name.

Examples

```
LOGICAL <DK    >, <DUA1  >       ;For I/O system (ACP)

LOGICAL MYLINE, <TTA2  >         ;For I/O system (ACP)

LOGICAL <PIPE  >, <RBUF3 >       ;Logical to structure name

LOGICAL REMNOD, <5.111  >        ;For DECnet communications (NSP)
```

4.3.8 MEMORY Macro

The set of MEMORY macros included in a configuration file initializes the kernel's memory configuration table, which is also used by the MIB utility in building the memory image file. A given MEMORY macro invocation describes the origin, size, type, and characteristics of one continuous segment of memory in the target system. You use the MEMORY macro once for each noncontiguous segment of memory and/or once for each memory segment that differs from a neighboring segment in type or characteristics. You can also use the MEMORY macro to reserve and name a segment of memory that is not to be allocated to any static process but that will be dynamically accessible to several processes as a physical shared region. For example, you might use this optional feature of the macro to reserve for shared access the area of memory associated with a bit-mapped-graphics controller board. See the ACSR$ primitive description in Chapter 3 and the discussion of shared regions in Chapter 7.

Note

> The MEMORY macro can be specified a maximum of 168 times in a configuration file.

To describe your target memory, you need to know the following specifics:

- The type of memory (ROM or RAM) assigned to a given range of addresses
- The base of that memory segment in 64-byte address units
- The size of the segment in 64-byte increments
- The presence or absence of memory parity checking
- The volatile or nonvolatile characteristic of the segment if it is RAM

If you specify memory parity checking for any memory segment, the MPT trap handler is automatically included in the kernel. The MPT handler raises an ES$MPT exception condition for any memory parity error.

If you describe one or more RAM segments as nonvolatile (implemented with battery backup), a special warm-restart module is included in the kernel to inhibit the standard reinitialization of user data areas on a restart with respect to data located in a nonvolatile RAM segment. Warm restart is intended to support user-implemented data recovery across power failures; see the PWFL$ primitive in Chapter 3 and the debugging considerations described at the end of this section.

Note

> In a KXT11–CA target system, the last two words of native RAM are used for power-failure flags, and the 60 bytes before that are used by native firmware. Therefore, the highest 64-byte unit of native RAM must not be included in the size specified in the MEMORY macro for that segment.
>
> In a KXJ11–CA target system, the addresses 157600(8) to 157777(8) are used for the native firmware stack on powerup. Therefore, these addresses should not be specified in any memory macros. See the files CFDKJU.MAC and CFDKJJ.MAC.

Syntax

$$\text{MEMORY base=mmmmm, size=nnnnn, type=}\left\{\begin{array}{c}\text{RAM}\\\text{ROM}\end{array}\right\}\text{, parity=}\left\{\begin{array}{c}\text{YES}\\\text{NO}\end{array}\right\}\text{,}$$

$$\text{[csr=addr], volatile=}\left\{\begin{array}{c}\text{YES}\\\text{NO}\end{array}\right\}\text{ [,res=}\left\{\begin{array}{c}\text{YES}\\\text{NO}\end{array}\right\}\text{, name=aaaaaa]}$$

Arguments

base=mmmmm

Specifies the base address of the memory segment divided by 100 if expressed in octal or by 64 if expressed in decimal. Default is 0.

size=nnnnn

Specifies the size of the memory segment, expressed as the number of 64-byte units in the segment. Thus, an 8K-byte segment size would be specified as 200 in octal (20000/100) or as 128 in decimal (8192/64).

type=

Identifies the type of memory segment: ROM or RAM. The default type is RAM.

parity=

Specifies whether the memory segment is parity checked. The default is NO.

csr=addr

Specifies the address of the control and status register (CSR) associated with parity checking of the memory segment. The value of this argument is meaningful only if parity=YES. Otherwise, the argument should be null.

volatile=

Describes a RAM memory segment as being either volatile or nonvolatile with respect to interruption of the normal power supply. Specify volatile=NO if the memory segment is provided with battery backup or is to be treated by the kernel as if it were nonvolatile for debugging purposes. (Recovery from a power failure can be at least partially tested by "faking" nonvolatile RAM and simulating power failures with INIT/RESTART commands while running an application under the PASDBG symbolic debugger.) Special debugging rules for an application described as having nonvolatile RAM are given at the end of this section.

The default is VOLATILE=YES, and the argument value is not meaningful for a ROM segment.

res=

Specifies whether the segment is to be reserved for run-time shared access. If res=YES, the described segment is not included in the memory to be allocated by MIB, and the kernel creates a shared region descriptor (SRD) for the segment at start-up time, identified by the name specified in the name argument.

The default is RES=NO.

name=aaaaaa

Specifies a 6-character ASCII string to be used as the run-time name of the shared region; that is, the name of the corresponding SRD. The name is padded with trailing blanks if it has fewer than six characters. This argument is meaningful and mandatory only if RES=YES. Otherwise, the argument should be null.

Restrictions

If multiple MEMORY macros are used in a configuration file, they must appear in ascending order of segment base addresses.

Do not describe any memory that cannot be allocated or is not to be reserved as a fixed-location shared region. Do not specify the memory space occupied by an ODT/bootstrap ROM chip, for example, any RAM that is used exclusively by processor firmware, or I/O page locations.

For an unmapped target system, the MIB utility makes a simple check to ensure that the memory image size defined by the MEMORY macro(s)—including any "holes" in the physical address space—does not equal or exceed 32K words. The MIB utility rejects any 32K words or larger memory image as an error. That limit makes no allowance for the I/O page, since MIB cannot determine the size of the I/O page. (The practical limit, of course, will be 28K words for an unmapped target with a 4K-word I/O page.)

Examples

```
MEMORY  BASE=0, SIZE=512., TYPE=ROM
MEMORY  BASE=1000, SIZE=384., TYPE=RAM, VOLATILE=NO
```

These calls describe a ROM segment consisting of 32,768 (512 * 64) bytes originating at location 0 and a RAM segment consisting of 24,576 bytes starting at location 100000—32768(decimal). Neither segment is parity checked, and the RAM segment is nonvolatile.

```
MEMORY  BASE=1000, SIZE=384., TYPE=RAM, VOLATILE=NO, RES=YES, NAME=SHRREG
```

The shared region (named SHRREG) consists of 24,576 bytes and originates at location 100000—32768(decimal). The region is not parity checked and is nonvolatile.

Note

> In DIGITAL-supplied configuration files, size values that are multiples of 32 (indicating 1K-word increments) such as 512 and 384 would appear as <16.*32.> and <12.*32.> , respectively. The latter are equivalent expressions intended to indicate 16K-word and 12K-word segment sizes, respectively, by notational convention.

Rules for Debugging an Application with Nonvolatile RAM

If you have described your target system as having nonvolatile RAM (VOLATILE=NO specified for one or more RAM segments), the following rules apply when debugging under PASDBG with regard to simulating a cold start or a power failure and a subsequent warm restart:

1. Whenever you load the target image with the PASDBG LOAD or LOAD/TARGET command, all user RAM is zeroed during the loading operation, and the kernel recognizes a cold start when you start the application. Subsequently, the PWFL$ primitive or Pascal POWER_FAIL function call will return a FALSE value, indicating a cold start. (On any start, cold or warm, the kernel initializes its own RAM data area, clearing all of system-common memory.)
2. After the application has been loaded and run, if you stop execution and issue the INIT/RESTART command without modifying the kernel flag word $PWFL1 as described below, the kernel recognizes a warm restart and clears only those user RAM segments declared as volatile, if any. (The effect of the INIT/RESTART in this case is to simulate a power failure and a following powerup, with retention of user read/write data located in RAM segments declared as nonvolatile—whether or not those segments are provided with battery backup.) Subsequently, the PWFL$ primitive or Pascal POWER_FAIL function call will return a TRUE value, indicating a warm restart.
3. After the application has been loaded and run, you can simulate a cold start without having to reload the application image by zeroing location $PWFL1 in kernel data space before issuing an INIT/RESTART command. (The effect of clearing location $PWFL1 prior to the restart is equivalent to an initial load or reload of the target.) If the kernel flag word $PWFL1, associated with the kernel's powerup processing, is clear before an INIT/RESTART, the kernel recognizes a cold start and sets its start/restart indicators accordingly.

In addition to the rules stated above, the following special rules apply only if you are debugging a KXT11-CA target system described as having nonvolatile RAM:

4. If you apply power to the target system just before loading the application image with a PASDBG LOAD or LOAD/TARGET command, the target's powerup firmware executes properly, and you need not take any special action with respect to the status CSR at 175002 controlled by that firmware.
5. If, after the application has been loaded and run, you want to either reload the application image or restart the application with an INIT/RESTART command, you must use ODT to open location 175002 in the I/O page and set bit 10 of that location, which is a CSR associated with the powerup firmware. This bit indicates that the system has powered up with battery backup. (Neither the LOAD nor the INIT/RESTART command of itself causes the powerup firmware to be entered.) You can then issue either the LOAD or the INIT/RESTART command to effect the kind of start or restart desired. (Setting of the CSR bit is independent of clearing or not clearing the kernel flag word $PWFL1 to condition a restart, as described in items 2 and 3 of the rules for all target systems.)

4.3.9 PRIMITIVES Macro

The PRIMITIVES macro determines the set of primitive service modules to be included in the kernel for your application. Primitives can be selected by means of the PRIMITIVES macro calls only if the optimize=YES option has been specified in the SYSTEM macro. If the optimize=NO option is specified or defaulted, the default value for the PRIMITIVES macro, ALL, is assumed, and all primitive service modules are included in the kernel.

If optimize=YES is specified in the SYSTEM macro, primitives can be selected in several ways:

- By classes—groups of functionally related primitives—identified by a functional class name, such as BCSEM for the group of binary and counting semaphore primitives
- By the special class name V1, implying all MicroPower/Pascal Version 1 primitives
- By individual primitive names—the first four characters of the corresponding primitive-request macro name, such as SGNL and WAIT
- By the single parameter ALL, implying all primitives

Up to six parameters may be specified in a given invocation of the PRIMITIVES macro, and the macro may be invoked as many times as needed if any of the first three selection methods are used.

Note

If you use the special "repetitive merge" method of optimizing primitive modules, which involves the use of the MERGE utility's auxiliary file feature, do not include any PRIMITIVE macros in the configuration file when building the kernel. Do specify optimize=YES in the SYSTEM macro, however. See the MicroPower/Pascal system user's guide for your host system for information on optimizing the kernel with respect to primitive modules by means of the "repetitive merge with auxiliary file" method.

Syntax

```
PRIMITIVES p1,p2,p3,p4,p5,p6
```

Parameters

p1 to p6

The possible values for p1 through p6 are as follows:

ALL	All currently available primitives
BCSEM	The binary and counting semaphore primitives SALL, SGLC, SGNL, WAIC, and WAIT
COMPLX	The complex primitives GELA, RCVA, WAIA, and WAQA and the related SLEP timer primitive
DRAM	The dynamic RAM allocation, sharing, and mapping primitives ACSR, ALRG, CRSR, DLRG, GMAP, MAPW, RCTX, SCTX, and UMAP
EXCMGT	The exception-management primitives CCND, DEXC, REXC, and SERA
INTMGT	The interrupt-management primitives CINT, DINT, and SPL
LOGNAM	The logical-name primitives CRLN and TRLN
PRMGT	The process-management primitives CHGP, CRPC, DLPC, GTST, RSUM, SCHD, SPND, STPC, and SSFA
QSEMN	The nonprivileged, high-level queue semaphore primitives RCVC, RCVD, SNDC, and SEND
QSEMP	The privileged, low-level queue semaphore primitives ALPC, ALPK, DAPK, SGLQ, SGQC, WAIQ, and WAQC
RBUF	The ring buffer primitives GELC, GELM, PELC, PELM, and RBUF
STRMGT	The structure-management primitives CRST, DLST, and GVAL
TIMER	The clock services primitives GTIM, SLEP, and STIM
V1	All primitives that were available in Version 1 of MicroPower/Pascal
xxxx	An individual primitive name, as represented by the first four characters of the corresponding MACRO–11 primitive service call, omitting the trailing $ character

The ALL value is mutually exclusive of other parameter values and when explicitly specified it must be p1. The parameter values for class name and single primitive name may be intermixed as desired. The default is ALL.

Examples

```
PRIMITIVES V1,TIMER,DRAM   ;Include specified primitive classes

PRIMITIVES ALPC,CRST,CHGP,DAPK,SGLQ,WAIQ   ;Include named primitives
```

4.3.10 PROCESSOR Macro

The PROCESSOR macro describes the type of processor used in the target system and optional hardware features. The processor type may be any of the following:

- LSI–11/2 (L112)
- LSI–11/23 (L1123)

- SBC–11/21 FALCON (FALC)
- SBC–11/21 FALCON–PLUS (FALCPLUS)
- KXT11–CA (KXT11C)
- KXJ11–CA (KXJ11C) (Specified separately from other J11-based processors)
- CMR21 (CMR21)
- J11-based processor such as an LSI–11/73, PDP–11/83 (J11), or MicroPDP–11/53 (but not a KXJ11–CA, which is specified separately)

Your target configuration may include a memory-management unit, a floating-point instruction option (FP–11, FIS, or FPA), and a fixed-frequency system clock. The PROCESSOR macro specifies those features if they are present and are used by the application. The PROCESSOR macro must appear and should be the third macro in the configuration file, immediately following the SYSTEM macro.

If the PROCESSOR macro specifies a memory-management unit (MMU=YES), mapping support and an MMU trap handler are included in the kernel. (A target system that has a memory-management unit can be used in unmapped mode, if desired.)

Note

For a J11-based target system (other than a KXJ11–CA), you can specify TYPE=J11, MMU=YES, and J11MAP=NO to operate the target as a mapped system without J11-specific capabilities (for example, without I&D-space separation or supervisor mode). Alternatively, specifying TYPE=L1123 and MMU=YES produces a functionally equivalent kernel. A setup without J11-specific capabilities is desirable if the user static processes do not exceed 56KB in size and the application does not require a supervisor-mode shared library. The advantage is minimized kernel overhead: faster context switching and smaller PCBs, primarily. Specification of TYPE=J11 and MMU=YES implies full J11 capabilities. (If MMU=NO, then TYPE=J11 and TYPE=L1123 are functionally equivalent, generating a kernel for an unmapped target. Usable physical memory cannot exceed 56KB.)

For a KXJ11–CA (TYPE=KXJ11C), you can also operate the target as a mapped system without J11-specific capabilities by specifying J11MAP=NO.

If the PROCESSOR macro specifies that the target supports the FP–11 or the FIS floating-point instruction set, the appropriate trap handler is included in the kernel. Note that the KEF11 and FPF11 hardware options are equivalent for configuration purposes; both support the FP–11 instruction set.

If the PROCESSOR macro specifies that the target (for example, a PDP–11/83) has a floating-point accelerator (FPA), the appropriate trap handler is included in the kernel. The included code correctly handles the asynchronous floating-point exceptions that occur with FPAs.

Note

Enabling FPA support adds two instructions to the kernel common exit code. This may slightly increase kernel processing overhead.

If the PROCESSOR macro specifies a clock, in terms of its frequency and its CSR, if any, the appropriate clock-handling service routine is included in the kernel. The clock argument should be specified if the application uses the kernel sleep primitive, the complex primitive timeout option, or network or point-to-point communication.

You can also indicate in the PROCESSOR macro where the effective interrupt vector area ends by specifying the first free address above the highest vector used on the target system. Doing so permits kernel code to begin at a location below the end of the standard vector area; that is, below address 1000 for an LSI-family or J11-based processor (other than a KXJ11–CA) and below 400 for an SBC–11/21, KXT11–CA, KXJ11–CA, or CMR21 processor. If the vector argument is not specified, the standard vector area for the target processor is assumed.

Syntax

PROCESSOR mmu, [fpu], type, j11map, vector, clock [,clkcsr]

Arguments

mmu

Specifies the effective presence (YES) or absence (NO) of a memory-management unit, determining whether the application is to be mapped or unmapped. (A memory-management unit might be physically present on the target but will be unused if mmu=NO.)

The default is MMU=NO.

fpu

Specifies the effective presence of the FP–11 floating-point option (fpu=FP11 or FPP), the FIS floating-point option (fpu=FIS), or a floating-point accelerator (fpu=FPA). If the target processor does not have floating-point hardware or it is not used by any user process, omit the fpu argument.

type

Identifies the type of target processor, as indicated by the parameter value. If the type is FALC or FALCPLUS, the FALCON macro must appear in the configuration file; if the type is KXT11C or KXJ11C, the KXT11C or KXJ11C macro, respectively, must appear. (Types L112 and L1123 are equivalent for configuration purposes; significant differences are indicated by the mmu, fpu, and clkcsr parameters. The distinction is provided for documentation only.)

The default is TYPE=LL123.

j11map

Indicates, for a J11-based processor or a KXJ11–CA, whether J11-specific capabilities (I&D-space separation and supervisor-mode) are to be supported. If the processor type is J11 or KXJ11C, the default is YES. For other processor types, this argument is ignored and is treated as a value of NO.

vector

Specifies the address of the next free location above the highest interrupt or trap vector configured on the target system. Effectively, the argument value indicates the point in memory at which the kernel's code may begin.

The default is 1000(octal) for an LSI- or J11-based target or 400(octal) for an SBC–11/21, KXT11–CA, KXJ11–CA, or CMR21 target.

clock

Indicates, by a frequency specification (nnHz), the effective presence of a source of clock interrupts through vector 100 or indicates the effective absence of clock source (NONE). (The kernel sleep primitive, the complex primitive timeout option, and system processes that provide network and point-to-point communications services require a system clock.) If a clock is specified here, you must also specify vector 100 in a DEVICES macro.

clkcsr

Specifies the address of the CSR associated with the clock, if any. (Clock CSRs are implemented on LSI–11/23–PLUS, KXT11–CA, KXJ11–CA, and J11-based targets and on the MXV11B multifunction board.)

If the target has no clock CSR or clock=NONE, omit the clkcsr argument. Where implemented, the standard clock CSR address is 177546 for all targets except the KXT11–CA and KXJ11–CA, for which the address is 177520.

Examples

```
PROCESSOR MMU=YES, FPU=FP11, TYPE=L1123, CLOCK=60HZ

PROCESSOR FPU=FIS, TYPE=L112, VECTOR=400

PROCESSOR MMU=YES, TYPE=J11, CLOCK=60HZ, CLKCSR=177546

PROCESSOR MMU=NO, TYPE=KXT11C, CLOCK=60HZ, CLKCSR=177520

PROCESSOR MMU=YES, TYPE=KXJ11C, CLOCK=60HZ, CLKCSR=177520

PROCESSOR TYPE=FALCPLUS, CLOCK=60HZ
```

4.3.11 RESOURCES Macro

The RESOURCES macro determines the amount of read/write memory (RAM) to be included in the kernel's data space for the following purposes:

- Message packets—the free-packet pool
- System data structures other than packets—the kernel free-memory pool
- The kernel-interrupt stack
- The free-RAM table, used by MIB during memory image building and by the kernel for creation of the run-time free-RAM list

Reasonable defaults are provided for the memory to be allocated for each of those uses, but you can override any or all of the default values by specifying optimize=YES in the SYSTEM macro and by including a RESOURCES macro in the configuration file. Note that RESOURCES can be invoked only if optimize=YES.

The free-packet pool and the free-memory pool form the kernel's system-common memory area. (Before using the packets or structures arguments of the RESOURCES macro, see Chapter 2 for information on system-common memory organization.) The default free-packet pool permits run-time allocation of 20 packets; thus, 20 packets are available for use by processes at the same time. That number has been found adequate for most applications and may be excessive for some. Since packets are reusable, the optimum number of packets is generally a space/performance tradeoff, but too few packets relative to application needs can cause obscure real-time problems. If space is not an overriding consideration, specify the maximum number of packets that could be required simultaneously by all processes in your application. (Implicit Pascal I/O support routines may use a number of packets.)

The default free-memory pool provides 3000 bytes for allocation of all other system data structures allocated at run time, such as PCBs, semaphores, and ring buffers (see Chapter 2 for individual structure sizes). The default pool size is sufficient for complex unmapped applications with many processes and structures and should be adequate for many mapped applications as well. Mapped applications require a larger free-memory pool than do unmapped applications—even assuming the same number of processes and structures—due in part to the mapping-image save/restore area associated with mapped PCBs (16 or 32 words for each process). In either case, if the pool is too small, a run-time failure will occur because of a process's inability to create another process or a needed data structure or the kernel's inability to create all static processes. If space is not a constraint, start with 4K or more bytes for the free-memory pool for a large mapped application; the pool size can be reduced in a later build cycle if experience shows it to be excessive. (While debugging, you can use the SHOW FREE STRUCTURES command to assess the amount of unused pool space at various points of system operation. Correspondingly, the SHOW FREE PACKETS command shows the number of free packets at any given point.)

The default kernel-interrupt stack size is a "safe" value and is sufficient for virtually all applications. Additional kernel stack might be needed if the application software includes—in addition to three standard device drivers of different hardware priorities—a user-written interrupt service routine that pushes more than eight words on the stack or a FORK routine that pushes more than nine words on the stack. In any case, you should allocate additional stack space only if you encounter kernel-stack overflow problems at run time. You can determine whether a kernel-stack overflow has occurred by examining the guardword at kernel location $KSTKL. If overflow has occurred, the guardword will contain a value other than the preset octal value 42557—defined by the kernel symbol $GRD.

MIB uses the free-RAM table to keep track of unallocated RAM memory areas during image building. The table is retained in the memory image as part of kernel read-only data and is used to form the run-time free-RAM list for dynamic RAM allocation. The default table size allows entries for five noncontiguous areas of free RAM, ordinarily a quite "safe" value. At build time, overflow of the table that is due to a very fragmented target memory configuration would be indicated by the MIB warning message "?MIB-W-Kernel Free RAM table too fragmented." Some memory space might be wasted if the table overflows; that is, one or more otherwise usable "holes" could exist in the resulting memory image. You can increase the table size in 4-byte increments to allow for additional entries if needed.

Some memory in addition to that specified by RESOURCES is included in the kernel's impure area for its private data structures and for interrupt dispatch blocks (IDBs). The amount of space allocated for IDBs depends on the number of vectors specified in the DEVICES macro; see Chapter 7 for details.

Syntax

```
RESOURCES stack=nnn, packets=nn, structures=nnnn, ramtbl=nn
```

Arguments

stack

The size in bytes of the kernel interrupt stack. The default size is supplied by the kernel symbol ..KIS. The recommended method of increasing the stack size, if necessary, is to specify the symbol ..KIS plus an increment, in an expression of the form <..KIS+n> . (The default size could change in a future version; use of the symbol guards against a potential version skew.)

packets

The number of message packets that can be allocated concurrently from the free-packet pool. The default is 20 packets, implying a free-packet pool of 800 bytes.

structures

The size in bytes of the free-memory pool, from which all dynamic system data structures are allocated. The default is 3000 bytes.

ramtbl

The size in bytes of the kernel free-RAM table. You can use this argument to change the number of 4-byte entries that can be accommodated. The default size is 20 bytes (five entries), which is excessive for a target system configured with one or two continuous segments of RAM. If the argument is specified, the value must be a multiple of 4.

Example

```
RESOURCES PACKETS=30., STRUCTURES=4096.
```

This macro statement allocates space for 30 message packets (1200 bytes) and 4K bytes of memory for other dynamic structures, implying a fairly large, complex application. The macro also allocates a standard-size kernel interrupt stack and a 20-byte free-RAM table by default.

4.3.12 SYSTEM Macro

The SYSTEM macro determines whether debugging support is included in the kernel, whether certain kernel-configuration parameters are to be defaulted or specified in the configuration file, whether address checking should be performed by some primitive operations, whether the system can be network booted, and whether the system will respond to a network request to reboot. The SYSTEM macro must appear and should be the second macro in the configuration file, immediately preceding the PROCESSOR macro.

The five SYSTEM macro arguments (optimize, debug, addrcheck, netboot, and nettrigger) take YES or NO values. If you specify NO for the optimize argument, the SYSTEM macro supplies default values for the RESOURCES, PRIMITIVES, and TRAPS macros, and those three macros may not appear in the configuration file. Effectively, optimize=NO produces a standard kernel-software configuration, with default system-common memory resources, all primitive service modules, and a basic set of trap processors appropriate to the target system. (Certain options expressed in other macros, such as mmu and fpu in the PROCESSOR macro, imply the presence or absence of optional trap processors.)

If you specify YES for the optimize argument, the default values for the RESOURCES, PRIMITIVES, and TRAP macros are inhibited, and you use those macros to tailor the kernel to your specific application requirements. The RESOURCES macro lets you specify the amount of RAM to be included in the kernel's data space for various purposes. The PRIMITIVES and TRAPS macros let you specify the primitive service modules and the set of trap processors to be included in the kernel. (If you use the special "repetitive MERGE with auxiliary files" method of optimizing the kernel with respect to primitive modules, described in the MicroPower/Pascal system user's guide for your host system, you must omit the PRIMITIVES macro from the configuration file for the build cycle in which that form of optimization is performed.)

If you specify YES for the debug argument, the debugger service module (DSM) is included in the kernel. Include the DSM only if you intend to use the PASDBG symbolic debugger to load and debug the application. If you do not plan to use PASDBG or will use PASDBG only for loading (LOAD/EXIT command), the DSM must not be included in the kernel (debug=NO). If the DSM is present, it automatically gains control at start-up time; that would cause a system hang in a nondebug target configuration or if PASDBG LOAD/EXIT were used.

If you specify YES for the addrcheck argument, the kernel performs some checks on primitive-request address parameters that are normally appropriate only during the debugging phase of application development. (The option controls occurrences of the ES$IAD—invalid address—exception return for some but not all primitive operation.) If you do not specify the addrcheck argument, it defaults to the value of the debug argument.

Note

> If you specify optimize=YES but do not include the RESOURCES macro in the configuration file, an assembly error is generated. Presence of the PRIMITIVES and TRAPS macros is not enforced, however.

If you specify YES for the netboot argument, the application image can be downline loaded and booted from an RSX or VMS host via DECnet/Ethernet. Ethernet downline loading is for debugged applications only (debug=NO). If netboot is enabled and the application is mapped, information supplied by the host (NCP)—such as the target node number—is used by MicroPower/Pascal network services, if present in the application. Potentially, the same application image can be loaded onto multiple machines in a network. If you do not specify the netboot argument, it defaults to NO, indicating that a different loading method is used.

If you specify YES for the nettrigger argument, the target system responds to a network request to reboot after the application is up and running. The Ethernet (QN) driver must be present in the application. If nettrigger is enabled, host system NCP commands can be issued to reload the same application image or to load a new image onto the target system. If you do not specify the nettrigger argument, it defaults to NO, disabling the netboot capability.

Syntax

$$\text{SYSTEM optimize=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}\text{, debug=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}\text{, addrcheck=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}\text{,}$$

$$\text{netboot=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}\text{, nettrigger=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}$$

Arguments

optimize=

Specifies whether you want to optimize the kernel by using additional configuration macros (YES) or want the default kernel configuration (NO).

debug=

Specifies whether the debugger service module (DSM) should be included in the kernel. If the DSM is included, it preempts use of the console-terminal serial line (vectors 60 and 64 and CSR 176560), which is required by PASDBG.

The default is no DSM.

addrcheck=

Specifies whether you want to receive address-checking (ES$IAD) error/exception returns for invalid buffer or data structure addresses specified in certain primitive service requests. (The individual primitive descriptions in Chapter 3 identify the error returns that are affected by this option.) The default is NO.

netboot=

Specifies whether the system can be network booted. The default is NO.

nettrigger=

Specifies whether the system responds to a network request to reboot. The default is NO.

Example

```
SYSTEM OPTIMIZE=NO, DEBUG=YES
```

The ADDRCHECK option defaults to YES because of the DEBUG option value. The NETBOOT and NETTRIGGER options default to NO.

4.3.13 TRAPS Macro

The TRAPS macro defines the set of trap-processing modules to be included in the kernel in addition to those implied by hardware-configuration parameters in other macros. (Use this macro only if the SYSTEM macro specifies optimize=YES.) You need to include in the kernel the trap handlers for the kinds of traps that your application may generate. If a given trap handler is not included and the corresponding trap occurs, the trap is effectively ignored.

The trap handlers needed for a given application depend on the hardware characteristics of the target with respect to error traps, such as illegal-instruction and nonexistent-memory faults, and on whether the application uses any explicit "service" trap instructions (EMT, TRAP, or BPT). The possible range of error traps varies according to both the type of processor and hardware options. (For example, an SBC–11/21 processor does not generate a trap-to-4 but may generate a break or NXM trap to vector 140 that the other processors do not.)

Certain options expressed in macros other than TRAPS cause implicit inclusion of appropriate trap handlers, as follows:

- PROCESSOR macro—The MMU=YES option implies the memory-management unit (MMU) trap handler, and the FPU=FP11 (or FPP), FPU=FIS, or FPU=FPA implies the corresponding floating-point trap handler.

- MEMORY macro—The PARITY=YES option implies the memory-parity error (MPT) trap handler.
- FALCON macro—The BREAK=EXCEPTION option implies the break (BRK) trap handler.

The TRAPS macro default of ALL causes inclusion of the basic set of error and service trap handlers that are applicable to the target processor. Thus, the TRAPS ALL default is generally appropriate unless you want to exclude handlers for EMT, TRP, and/or BPT traps. (The BPT handler is needed only for user-coded BPT instructions, not for breakpoints set by means of PASDBG.) If the SYSTEM macro specifies optimize=NO, the TRAPS ALL default is assumed.

Syntax

TRAPS [t1,t2,t3,t4,t5,t6,t7,t8]

Arguments

tn

May be either the mnemonic for one of the trap-processing modules listed below or ALL (default). Up to eight arguments may be specified in each macro call.

The trap handler mnemonics are:

BPT	Breakpoint instruction trap
BRK	SBC–11/21 break-character exception trap
EMT	EMT instruction trap
FIS	FIS exception trap
FPA	FPA exception trap
FPP	FP11 exception trap
MMU	Memory-management fault
MPT	Memory parity error
T10	Trap to 10
TR4	Trap to 4 (invalid for SBC–11/21 target)
TRP	TRAP instruction trap

For all processor types other than FALC and FALCPLUS, the basic set of trap handlers implied by TRAPS ALL is TR4, T10, BPT, EMT, and TRP.

For processor type FALC or FALCPLUS, the basic set of trap handlers implied by TRAPS ALL is T10, EMT, and TRP. In addition, if the FALCON macro does not specify BREAK=ROMODT or BREAK=SFWODT, the BPT handler is implied, and NXM errors simulate a trap-to-4 exception (ES$BUS).

Example

```
TRAPS  TR4,T10,EMT       ;Excludes TRP and BPT from basic LSI set
```

Chapter 5

Dynamic RAM Allocation and Region Sharing

This chapter discusses the 10 primitive services that, collectively, let user processes do the following:

- Obtain an area of unused memory and, optionally, release it after temporary use (dynamic RAM allocation/deallocation)
- Share an area of "static" or "dynamic" memory between static process families (region sharing)
- In a mapped system, obtain a virtual-address window into either a dynamic or shared memory area, in support of capabilities 1 and 2 (dynamic mapping)

These related capabilities are intended principally to support large memory configurations and shared common memory in mapped target systems and are described here in that context unless otherwise indicated. Dynamic RAM allocation may be useful in some unmapped applications, but memory sharing by means of kernel services has a very limited utility, since more efficient design alternatives exist in the unmapped environment. All user processes can be members of the same static-process family, for example, since mapping considerations do not apply.

The Pascal and MACRO–11 primitive service requests corresponding to the capabilities previously listed are:

- Dynamic RAM allocation
  - ALLOCATE_REGION function; ALRG$ macro call
  - DEALLOCATE_REGION procedure; DLRG$ macro call
- Region sharing
  - CREATE_SHARED_REGION procedure; CRSR$ macro call
  - ACCESS_SHARED_REGION procedure; ACSR$ macro call
  - DELETE_SHARED_REGION procedure; DLSR$ macro call

- Dynamic mapping
  - MAP_WINDOW procedure; MAPW$ macro call
  - UNMAP_WINDOW procedure; UMAP$ macro call
  - GET_MAPPING procedure; GMAP$ macro call
  - SAVE_CONTEXT procedure; SCTX$ macro call
  - RESTORE_CONTEXT procedure; RCTX$ macro call

Chapter 18 of the *MicroPower/Pascal Language Guide* (for Pascal programmers) and Chapter 3 of this manual (for MACRO–11 programmers) describe the full syntax and formal semantics of the individual service requests. This chapter discusses and shows by coding examples the relationship between those requests. For your convenience, brief syntax summaries of the Pascal calls are provided where relevant.

5.1 Definition of Terms

Several terms are used here and elsewhere in this manual in a special, limited sense when referring to region allocation and sharing. The following paragraphs define those terms for a mapped-memory environment. Differences for the unmapped case are given at the end of the section.

Free RAM is that portion of the described RAM configuration that remains unused in the application memory image following a build cycle, as opposed to statically allocated RAM. Free RAM, if present, may consist of one continuous segment or several disjoint segments.

A **region** is a contiguous block of physical memory, normally RAM, and can be either a physical region or a common region.

A **physical region** is dynamically allocated from free RAM. The maximum size of a physical region is limited only by the size of the largest free-RAM segment available in an application. The granularity of both the physical base address and the size of a physical region is 32 words, or 100(octal) bytes, the size of a physical memory block for mapping purposes. (For brevity, this size unit is referred to as a "PAR tick.")

A **common region** is defined in one process's statically allocated memory space—typically but not necessarily RAM—for the purpose of memory sharing among several static processes. (The concept of a common region is relevant only in relationship to shared regions and, for the most part, only in mapped systems.) The maximum size of a common region is limited by virtual-address space considerations and is generally less than 24K words except in a system with I&D-space separation in effect. The granularity of size of a common region is effectively 32 words. A common region need not begin on a 32-word physical boundary.

A **shared region** is accessible by any and all processes in an application. A shared region has a run-time name that is assigned by the creator of the shared region and used by accessing processes in other static-process families. A named kernel structure, called a shared region descriptor (SRD), identifies a shared region as such. The mode of a shared region can be either physical or common.

A **region ID block** (RIB) is a 4-word data structure in the user's address space and is used to describe a region or a shared region.

A **window** is a sequence of virtual addresses in a program's address space having a 1-to-1 correspondence with a range of physical addresses in a region. The maximum size of a window is essentially limited by the "unused" virtual address space available to the accessing process, that is, by the number of free APRs available to the process for modification.

The differences to the preceding definitions for an unmapped-memory environment are as follows:

- Physical region—The granularity of size of a physical region is four bytes, and the region may start on any word boundary.
- Common region—The size of a common region has no fixed granularity; the size may be any number of bytes.
- Window—The concept is inapplicable.

5.2 Region ID Block (RIB)

The region ID block, or RIB, is a predefined MicroPower/Pascal data structure that you declare in a program. The RIB is a link between most of the primitive operations covered in this chapter, as discussed in Section 5.2.2.

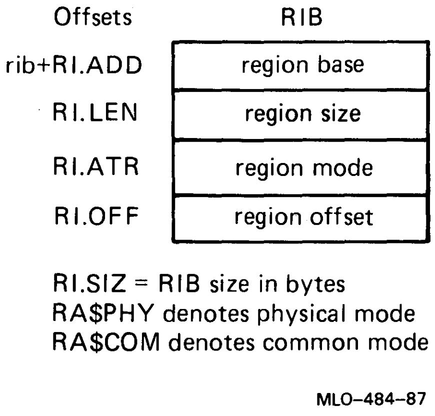
5.2.1 RIB Definition

The following type definition is provided in DRAM.PAS for declaring RIB variables in Pascal:

```
TYPE
  ADDRESS_TYPE = (COMMON,PHYSICAL);

  REGION_ID_BLOCK = RECORD
                    REGION_ADDRESS : UNIVERSAL;
                    REGION_SIZE : UNSIGNED;
                    REGION_MODE : ADDRESS_TYPE;
                    REGION_OFFSET : UNSIGNED;
                    END;
```

(The MACRO-11 region-mode symbols RA$COM [value=0] and RA$PHY [value=1] correspond to the Pascal constants COMMON and PHYSICAL of the type ADDRESS_TYPE.)

The MACRO–11 programmer may declare a RIB area with a .BLKW 4 directive or, more generally, with a .EVEN and .BLKB RI.SIZ directive and can use the predefined symbolic offsets shown in the following diagram:



Offsets	RIB
rib+RI.ADD	region base
RI.LEN	region size
RI.ATR	region mode
RI.OFF	region offset

RI.SIZ = RIB size in bytes
RA$PHY denotes physical mode
RA$COM denotes common mode

The RI.xxx and RA$xxx symbols are defined by the RIBDF$ macro in the MicroPower/Pascal COMM and COMU system macro libraries.

5.2.2 Relationship to Primitive Operations

A RIB address is required for the Allocate Region, Deallocate Region, Create Shared Region, Access Shared Region, and Map Window primitive calls. The RIB is a destination variable for the Allocate Region and Access Shared Region operations and a source variable for the Create Shared Region, Map Window, and Deallocate Region operations. In only one case, preparatory to creation of a shared common region, does the user program place any information in a RIB directly.

In a little more detail, the RIB's role in the various operations is:

- In dynamic RAM allocation/deallocation:
  - The Allocate Region primitive sets up a physical description of the allocated region in the caller's RIB. The region mode is set to PHYSICAL (RA$PHY). The RIB can then serve as input to Map Window and/or Create Shared Region and eventually to Deallocate Region.
  - The Deallocate Region primitive obtains the description of the region to be deallocated from the caller's RIB. (The RIB content is invalidated by the operation.)
- In common-region definition:
  - The user program "allocates," or defines, a common region by initializing a RIB with the description—virtual address, length in bytes, and mode—of the static data structure to be made shareable. The data structure might be, for example, a large array defined in the creating program's address space. The mode must be set as COMMON (RA$COM). A RIB so initialized is useful only as input to a Create Shared Region operation.

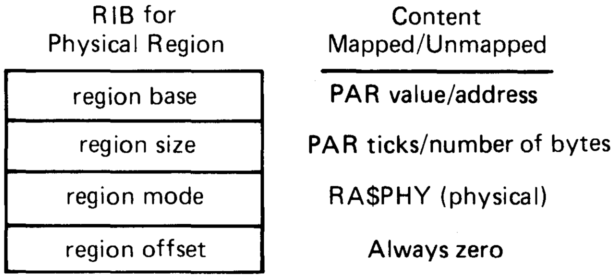
- In shared-region operations:
  - The Create Shared Region primitive obtains the description of the region to be made shareable—accessible to other processes—from the caller's RIB. The region mode may be COMMON (RA$COM) or PHYSICAL (RA$PHY). The primitive uses the information in the RIB, either "as is" for a physical region or transformed for a common region, to construct a shared region descriptor (SRD) in kernel space.
  - The Access Shared Region primitive sets up the physical description and mode of the named shared region, common or physical, in the caller's RIB. (The access request is made by a process other than the shared-region creator, of course.) The RIB can then serve as input to Map Window.
- In mapping and remapping operations:
  - The Map Window primitive obtains from the caller's RIB a physical description of the region to be mapped into. The information in the RIB was presumably set up either by Allocate Region for a physical region or by Access Shared Region for either a shared physical or shared common region. (Although any shared region is "physical" in relation to the accessing process—in the sense that it is not in the accessor's static virtual space—the COMMON/PHYSICAL mode distinction is critical to the operation of the Map Window primitive.)

5.2.3 Form and Use of RIB Content

The kind of information in a RIB and, consequently, the way it is used differs between mapped and unmapped environments. Also, in the mapped case, the RIB content for a shared common region differs somewhat from that for a physical region, depending on its point of use.

5.2.3.1 For Physical Regions

The following diagram illustrates the difference in RIB content for a physical region in mapped and unmapped usage:



RIB for Physical Region	Content Mapped/Unmapped
region base	PAR value/address
region size	PAR ticks/number of bytes
region mode	RA$PHY (physical)
region offset	Always zero

MLO–485–87

In mapped usage, the user code never modifies or uses RIB content directly. The Map Window primitive ultimately supplies the virtual pointer that the user program needs for references to the region. In unmapped usage, the user program normally does not modify RIB content but uses the address and size information set up by the Allocate Region or Access Shared Region primitive for references to the region.

5.2.3.2 For Mapped Shared Common Regions

The following diagram illustrates how the RIB content for a mapped common region varies, depending on its point of use:

RIB for Mapped Common Region	As Set Up by User Code	As Returned by Access Shared Region
region base	Virtual address	Physical PAR value
region size	Number of bytes	Number of PAR ticks
region mode	RA$COM (COMMON)	RA$COM (COMMON)
region offset	Must be zero	Bytes from PAR base (range: 0 to 31)

MLO-486-87

The user program that "creates" a shared common region sets up the base, size, and mode fields of the RIB for input to the Create Shared Region primitive. (The offset field is not significant at this point.) A user program that accesses a shared common region does not either modify or use RIB content directly but passes the RIB as input to Map Window.

In unmapped usage, the content of a RIB for a shared common region is always the same as that for a physical region except for the mode attribute, and the accessing program uses the region base and size fields directly for reference to the region. (The offset field is not significant for an unmapped common region.)

5.3 Dynamic Region Allocation and Use

The Pascal programmer can think of the combined Allocate Region and Map Window operations as analogous to a NEW operation and of the Unmap Window and Deallocate Region operations as analogous to a DISPOSE. The pointer variable obtained by means of Map Window is used in a very similar fashion to that obtained with NEW. The difference is that the memory space so obtained comes out of unallocated, unmapped physical memory, as opposed to memory statically allocated to the program's heap—to which the process is already mapped. A concomitant difference is that the process (static or dynamic) must have at least one APR available for dynamic modification. If you use Map Window in the standard, "safe" fashion described in this section, at least one full page (4KW) of the virtual addresses potentially available to the program (that is, to the static process family) has not been allocated at build time.

A process can obtain a physical region to:

- Acquire a relatively large "chunk" of physical memory on a temporary basis for either 1-shot or very intermittent use. In this case, the process deallocates the region after each such use. (The entire region may or may not fit in the process's unused virtual-address space; if it does not, iterative window mapping is required.) This usage assumes that, by convention, some or all of free RAM is treated as a serially reusable resource by two or more processes, with no functional connection between those processes.

- Circumvent the virtual-address space limitation of 32K words of code and data (or 32K of data, assuming I&D-space separation) in order to manipulate one or more large physical data structures. In this case, the dynamically acquired region or regions will presumably not be returned, and the process will typically devote one or more APRs—one or more 4K-word pages of virtual address space—for iterative window mapping. A physical region obtained for this reason is often also shared with another related process.

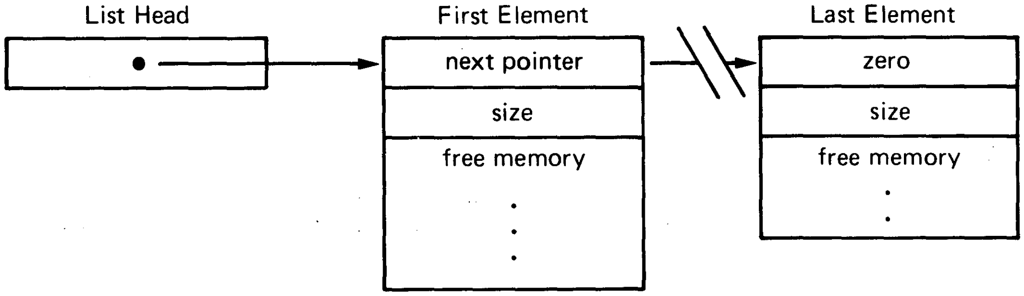
The next section describes how the kernel keeps track of and allocates free RAM.

5.3.1 Free-RAM List

The kernel maintains a linked list of free-RAM segments in unallocated memory, where each free segment is an element of the list, as illustrated in Figure 5–1. The list represents any read/write memory that was described in the system configuration file and not allocated to any application component at build time, less any currently allocated region(s). (The free RAM available at system-initialization time is identified as "Available RAM" in a memory-image map of the complete application produced by the MIB utility program.) Ordinarily, the initial free-RAM list contains a single segment if all of the target's RAM is physically continuous (no gaps or intervening ROM segments). However, free RAM could become fragmented during the build cycle because of the use of special RELOC and/or MIB utility options that "force" arbitrary, user-specified physical start locations.

The free-RAM list is in ascending-address order. Regions are allocated on a first-fit basis; that is, the required amount of memory is "removed" from the first-encountered list element that equals or exceeds the size of the requested region. On deallocation, the returned region is recombined with adjoining free segments, if any. The kernel does not keep track of a memory block once it has been allocated. The application program must return an allocated block when done with it, especially in the case of a terminating process.

Figure 5–1: Linked List of Free-RAM Segments



(The pointer and size values indicate PAR ticks in a mapped system or bytes in an unmapped system.)

MLO–487–87

5.3.2 Creation of Shared Regions at Build Time

Shared regions can be created dynamically at run time, or they can be set up at build time by means of the MEMORY macro. If they are set up at build time, the kernel allocates the regions and creates the shared regions at system start-up time. See the description of the MEMORY macro in Chapter 4 for more information.

5.3.3 Syntax of Relevant Pascal Requests

An informal presentation of the Pascal primitive request syntax is given here to minimize the need for reference to the *MicroPower/Pascal Language Guide* when reading examples in this chapter. See Chapter 3 of this manual for the corresponding MACRO–11 ALRG$, MAPW$, UNMAP$, and DLRG$ macro calls.

The syntax of the Allocate Region function call is as follows:

```
ALLOCATE_REGION (RIB := rib-variable, REG_SIZE := unsigned-value)
```

The REG_SIZE constant or variable parameter specifies the number of 64-byte units of memory, or PAR ticks, required; the value 128, for example, indicates a region size of 8192 bytes (4K words).

The essential syntax of the Map Window procedure call, omitting the optional STATUS parameter, is as follows:

```
MAP_WINDOW (ADDRESS_SPACE := space-option ,  {default D_SPACE}
            ACCESS := access-option ,  {default READ_WRITE}
            PAR_CHOICE := free/fixed-option ,  {default FREE}
            CACHING := caching-option ,  {default LEAVE}
            WINDOW_PTR := pointer-variable ,
            OFFSET := unsigned-value ,  {default 0}
            LENGTH := unsigned-value ,
            RIB := rib-variable )
```

The WINDOW_PTR parameter names a variable of type UNIVERSAL in which the procedure returns a beginning-of-window virtual-address value, which can subsequently be used as a pointer to a Pascal data structure, such as an array or a record. (The parameter is a destination-only variable when the PAR_CHOICE option is FREE, the default, representing the "safe" mode of using MAP_WINDOW.) The LENGTH variable or constant specifies a number of bytes, the length of the required window. The OFFSET variable or constant specifies a number of PAR ticks (64-byte units) and defaults to 0. The parameter is needed only if the required window is not to start at the beginning of the region described by the RIB. A description of the nondefault options for the ADDRESS_SPACE, ACCESS, PAR_CHOICE, and caching parameters is provided in Chapter 3. (ADDRESS_SPACE is significant only for target systems that support instruction- and address-space separation and only if such separation is in effect for the calling process.)

The essential syntax of the Unmap Window procedure call is as follows:

```
UNMAP_WINDOW (WINDOW_PTR := pointer-variable ,
              LENGTH := unsigned-value ,
              ADDRESS_SPACE := space-option )  {default D_SPACE}
```

The WINDOW_PTR parameter names the source variable containing the beginning-of-window address value of the window to be unmapped. Normally, this variable is of type pointer; the corresponding formal parameter is of type UNIVERSAL. The LENGTH variable or constant specifies the length of window to be unmapped, in bytes—essentially indicating the number of consecutive APRs to be made inactive. The nondefault option, I_SPACE, for the ADDRESS_SPACE parameter is described in Chapter 3. The purpose of the Unmap Window operation is to free up one or more APRs previously used to map a window so they can be reused in a subsequent Map Window operation. (Typically, MAP_WINDOW and UNMAP_WINDOW calls are used in symmetric pairs.)

The essential syntax of the Create Shared Region procedure call, omitting the optional STATUS parameter, is as follows:

```
CREATE_SHARED_REGION (RIB  := rib-variable ,
                      DESC := structure-descriptor,
                      NAME := region-name )
```

The specified RIB variable describes the region—physical or common—to be made shareable with any other process. The structure descriptor, if specified, receives the structure identifier of the shared region descriptor (SRD) that the kernel will create for the shared region. The region-name character string or variable of type NAME_STR provides the run-time name of the shared region.

The essential syntax of the Access Shared Region procedure call, omitting the optional STATUS parameter, is as follows:

```
ACCESS_SHARED_REGION (RIB  := rib-variable ,
                      DESC := structure-descriptor,
                      NAME := region-name )
```

The specified RIB variable receives a description of the region—physical or common—to be accessed. The structure descriptor, if specified, contains the structure identifier of the shared region descriptor (SRD) that was previously established for the shared region. (The DESC parameter must be omitted if the NAME parameter is supplied and vice-versa.) The region-name character string or variable of type NAME_STR specifies the run-time name of the shared region.

The syntax of the Deallocate Region procedure call is as follows:

```
DEALLOCATE_REGION (RIB := rib-variable )
```

The following coding examples illustrate the combined use of the several primitive services.

5.3.4 Coding Examples

In the examples in this section, the structure of the sample arrays and the nature of the operation performed on them are quite arbitrary and simplistic, sufficient to illustrate the programming principles involved without introducing excessive coding detail.

5.3.4.1 Unshared Use of a Physical Region

Assume that a process needs 32K bytes of memory in order to create and manipulate a rather large array and, consequently, would require four APRs if that much space were to be statically allocated to the program. Further assume that the array could not be accommodated in the process's virtual address space if it were statically allocated—declared as a program variable—so dynamic RAM allocation and mapping is needed. The minimum requirement is that the static-process family to which the example process belongs does not exceed 28K words, leaving at least one unused APR available for window mapping.

Keeping the stated assumptions in mind, consider the following program segment:

```
PROCESS  map1;

 1    %INCLUDE 'MICROPOWER$LIB:DRAM.PAS'    { To get the "right stuff" }

      TYPE
 2         big_record = ARRAY [0..1023] OF INTEGER;
 3         huge_record = ARRAY [0..4095] OF INTEGER;

      VAR
 4         i, off_set : INTEGER;
 5         p : ^ big_record;
 6         q : ^ huge_record;
 7         rib1 : REGION_ID_BLOCK;

      BEGIN
           { Allocate 32Kb of free RAM }
 8         IF NOT ALLOCATE_REGION (RIB := rib1, REG_SIZE := %O'1000')
                THEN  WRITELN ('Not enough free memory') {and quit}
                ELSE { go on with life hereafter }

      BEGIN
 9         FOR i := 0 TO 3 DO  { Repeat mapping with same PAR 4 times }
                BEGIN
10              off_set := i*128;  { Map one page at a time }
11              MAP_WINDOW (RIB := rib1, LENGTH := 8192,
                     OFFSET := off_set, WINDOW_PTR := q);
12              q^[0]      := (i*4);
13              q^[1024]   := (i*4) + 1;
14              q^[2048]   := (i*4) + 2;
15              q^[3072]   := (i*4) + 3;
16              UNMAP_WINDOW (WINDOW_PTR := q, LENGTH := 8192);
                END;

17         FOR i := 0 TO 15 DO
                BEGIN
                     { i*32 for 2048-byte records }
18              off_set := i * ( SIZE(big_record) DIV 64);
19              MAP_WINDOW (RIB := rib1, LENGTH := 2048,
                     offset := OFF_SET, WINDOW_PTR := p);
20              WRITELN (p^[0]);   { For program checkout }
21              IF p^[0] <> i
                     THEN  WRITELN ('Mapping error');
22              UNMAP_WINDOW (WINDOW_PTR := p, LENGTH := 2048);
                END;
           END;
      END;
```

The MAP1 example shows a single process that acquires, maps, and manipulates a physical region. (The process does not share the region with any other process.) The MAP1 process uses one available APR to repetitively map the entire 32KB region, in increments of four 8KB windows and sixteen 2KB windows, respectively. In statement 8, the process acquires the 32KB region whose description is returned by the primitive operation in the program variable rib1. The size of the requested region is specified in the allocation request as 1000(octal) PAR ticks, each tick consisting of 64 bytes.

In statements 9 through 16, the region is mapped incrementally in 8KB—128 PAR-tick—windows (full pages), and the first location in each 2KB "quadrant" of the windows is marked with an arbitrary value that is a function of the mapping loop index. Statements 17 through 22 remap the same region in 2KB increments, increasing the region offset by 2KB each time the region is mapped. The first location in each window is tested for a valid value.

In each mapping iteration, the UNMAP_WINDOW call frees the APR used for the window for subsequent reuse. The APR used for the mapping is chosen by the MAP_WINDOW primitive because the default value of FREE for the PAR_CHOICE option is in effect. The primitive uses the first—lowest-numbered—available APR for the mapping operation.

5.3.4.2 Shared Use of a Physical Region

Assume that two independent processes need to access the same 8K-byte area of memory, in the form of a 4K-array of integers. Further assume either that the array could not be accommodated in the "declaring" process's virtual address space if it were statically allocated— declared as a program variable—or that the array is needed only temporarily. Thus, dynamic RAM allocation, sharing, and mapping are indicated. The minimum requirement is that both of the static-process families to which the processes belong do not exceed 28K words, leaving at least one unused APR available for window mapping.

Keeping the stated assumptions in mind, consider the following two program segments:

```
     [SYSTEM (MICROPOWER)] PROGRAM map2;  { One static process }
1    %INCLUDE 'MICROPOWER$LIB:DRAM.PAS'   { To get the "right stuff" }
     TYPE
2         big_record = ARRAY [0..4095] OF INTEGER;
     VAR
3    p : ^ big_record;
4    ib1 : REGION_ID_BLOCK;
5    [INITIALIZE] PROCEDURE start;
     BEGIN
          { Allocate 8 Kb of free RAM }
6         IF NOT ALLOCATE_REGION (RIB := rib1, REG_SIZE := 128)
               THEN  WRITELN ('Not enough free memory');
7         CREATE_SHARED_REGION (RIB := rib1, NAME := 'SHARED');
8         MAP_WINDOW (RIB := rib1, LENGTH := 8192,
               OFFSET := 0, WINDOW_PTR := p);
9    p^[0] := 1;
     END;
     BEGIN
          .
          .
          .
```

```
10       WRITELN ('Map2 Done');
    END.

         { Another static process in the same application }
    [SYSTEM (MICROPOWER)] PROGRAM map3;

11  %INCLUDE 'MICROPOWER$LIB:DRAM.PAS'    { To get the "right stuff" }

    TYPE
12       big_record = ARRAY [0..4095] OF INTEGER;

    VAR
13       q : ^ big_record;
14       rib2 : REGION_ID_BLOCK;

    BEGIN
15       ACCESS_SHARED_REGION (RIB := rib2, NAME := 'SHARED');
16       MAP_WINDOW (RIB := rib2, LENGTH := 8192,
              OFFSET := 0, WINDOW_PTR := q);
17       IF q^[0] = 1
              THEN  WRITELN ('Successful mapping');
              ELSE  WRITELN ('Unsuccessful mapping');
                  .
                  .
                  .
     END;
```

The MAP2/MAP3 example shows two processes that interact through a shared physical region. The MAP2 process acquires a physical region and makes it shareable by any other process in the application. The MAP3 static process accesses the same region by means of its run-time name, "SHARED."

In statements 6 through 9, process MAP2 allocates an 8KB region (described as 128 PAR ticks), makes the region shareable, and then maps and "marks" the region with an initial value. All these operations are done within an INITIALIZE procedure so that the region will be available to the sharing process whenever that process starts up, regardless of the relative priorities of the two processes. In particular, statement 7 requests that the kernel create a system-wide data structure, called a shared region descriptor (SRD) and named SHARED, that describes the physical region described by the RIB "rib1".

In statement 15, the MAP3 static process obtains a physical description of the shared region in its RIB "rib2", by means of the SRD named SHARED. (Like all run-time names, the name SHARED is case sensitive.) In statement 16, process MAP3 maps the physical region described in rib2 and obtains a virtual-address (window) pointer to the corresponding array in the pointer variable q. The mapping operation uses the first—lowest-numbered—unused APR in the process's APR set for the window mapping. Statement 17 tests the array for its "initialization" value and reports the result before performing further, unstated operations on the array.

5.4 Shared Common Region Allocation and Use

To a process accessing a shared common region, that region looks the same as if it were a dynamic physical region. The process that owns the "region," however, initially declares it as a program variable of the required size and type and then defines the region as such by placing a description of the variable, in terms of its virtual location and size, in a region ID block (RIB). Thus, the region is in fact both an ordinary variable in the program's normally allocated address space and a memory area that can be made accessible to another process family through use of run-time kernel services.

A shared region should be deleted by the creating process if that process terminates, since the process's data space in which the corresponding variable exists is deallocated at process termination. (A variable of a dynamic process will not be deallocated if it is declared STATIC unless the parent static process also terminates, but the useful lifetime of a shared region as such must be coordinated between the sharing processes in any case.)

5.4.1 Syntax of Additional Relevant Pascal Request

The essential syntax of the Delete Shared Region procedure call, omitting the optional STATUS parameter, is as follows:

```
DELETE_SHARED_REGION (DESC := structure-descriptor )
```

or

```
DELETE_SHARED_REGION (NAME := region-name )
```

If specified, the structure descriptor contains the structure identifier of the shared region descriptor (SRD) that was previously established for the shared region. The region-name character string or variable of type NAME_STR specifies the run-time name of the shared region; the name of the corresponding SRD. Both forms of the request result in deletion of the identified SRD.

The following coding examples illustrate the combined use of the several primitive services.

5.4.2 Shared Common Region Coding Example

Assume that two independent processes need to access the same 2K-byte area of memory, in the form of a 1K-array of integers. Further assume that the array both can be accommodated in the declaring process's virtual-address space (that is, can be statically allocated as a program variable) and is needed throughout a run of the application. Thus, sharing and mapping of a common region are indicated. The minimum requirement is that the static-process family to which the accessing process in the example belongs does not exceed 28K words, leaving at least one unused APR available for window mapping.

Keeping the stated assumptions in mind, consider the following two program segments:

```
     [SYSTEM (MICROPOWER)] PROGRAM map4;  { One static process }
1    %INCLUDE 'MICROPOWER$LIB:DRAM.PAS'   { To get the "right stuff" }
     TYPE
2        biggish_record = ARRAY [0..1023] OF INTEGER;
```

```
    VAR
3        a : biggish_record;
4        rib1 : REGION_ID_BLOCK;
5        region_guard1 : SEMAPHORE_DESC;
6        region_complete1 : SEMAPHORE_DESC;

7   [INITIALIZE] PROCEDURE start;
    BEGIN
8        WITH rib1 DO
              BEGIN
9                  region_address := ADDRESS (a);
10                 region_size := SIZE (a);
11                 region_mode := COMMON;
              END;
         { Assume enough kernel data space for an SRD and several }
         {  semaphores; otherwise, fail with a RESOURCE exception }
12       CREATE_SHARED_REGION (RIB := rib1, NAME := 'COMMON');
13       a[0] := 1;
14       CREATE_BINARY_SEMAPHORE (DESC:= region_guard1,
                        VALUE := 1,   { Gate is open }
                        NAME := 'ACCESS');
15       CREATE_BINARY_SEMAPHORE (DESC:= region_complete1,
                        VALUE := 0,   { Gate is closed }
                        NAME := 'RGDONE');
    END;
    BEGIN
16       WAIT (DESC:= region_guard1);
             .
             .          { operations on region }
             .

17       SIGNAL (DESC:= region_guard1)
             .
             .          { other operations }
             .

18       WAIT (DESC:= region_complete1);
19       WRITELN ('Map4 Done');
20       (NAME := 'COMMON');
    END.

         { Another static process in the same application }
    [SYSTEM (MICROPOWER)] PROGRAM map5;

21  %INCLUDE 'MICROPOWER$LIB:DRAM.PAS'   { To get the "right stuff" }

    TYPE
22       biggish_record = ARRAY [0..1023] OF INTEGER;

    VAR
23       q : ^ biggish_record;
24       rib2 : REGION_ID_BLOCK;
25       region_guard2 : SEMAPHORE_DESC;
26       region_complete2 : SEMAPHORE_DESC;
```

```
    BEGIN
27       ACCESS_SHARED_REGION (RIB := rib2, NAME := 'COMMON');
28       MAP_WINDOW (RIB := rib2, LENGTH := 2048,
              OFFSET := 0, WINDOW_PTR := q);
29       IF q^[0] = 1
              THEN WRITELN ('Correct array access')
              ELSE BEGIN
                   WRITELN ('Incorrect array access');
                   STOP;
                   END;
30       INIT_STRUCTURE_DESC (DESC:= region_guard2,
                              NAME := 'ACCESS');
31       WAIT (DESC:= region_guard2)
              .
              .         { operations on region }
              .

32       SIGNAL (DESC := region_guard2)
              .
              .         { other operations }
              .
              { When finished with region }
33       SIGNAL (NAME := 'RGDONE')
    END.
```

The MAP4/MAP5 example shows two processes that interact through a shared common region. The MAP4 process declares a program variable, a, and makes it shareable with any other process in the application by describing the variable as a common region and then creating a shared common region from it. The MAP5 static process accesses the same region by means of its run-time name, "COMMON".

In statements 8 through 11, process MAP4 places the virtual description of the 2KB-array a in the RIB rib1, indicating that the memory area is a common region. Statement 12 creates the shared region descriptor (SRD)—a system data structure—with the run-time name "COMMON", based on the region description in the variable rib1, thus making the region shareable. Statement 13 "marks" the first element of the corresponding array with an initial value. All these operations are done within an INITIALIZE procedure so that the region will be available to the sharing process whenever that process starts up, regardless of the relative priorities of the two processes.

As a side effect, the virtual region description placed in variable rib1 by statements 9 and 10 is transformed by the Created Shared Region operation to a physical memory description that is directly usable in a mapping operation. (The transformed description is identical to that provided to process MAP5 by the Access Shared Region operation requested in statement 29.)

The static process MAP5 is structurally identical to process MAP3, previously described. In statement 27, the MAP5 static process obtains a physical description of the shared common region in its RIB "rib2", by means of the SRD named "COMMON". (Like all run-time names, the name COMMON is case sensitive.) In statement 28, process MAP5 maps the common region described in rib2 and obtains a virtual-address (window) pointer to the corresponding array in the pointer variable q. The mapping operation uses the first—lowest-numbered—unused APR in the process's APR set for the window mapping. Statement 29 tests the array for its "initialization" value and reports the result before performing further, unstated operations on the array.

5.4.3 Virtual Array Coding Example

The example in this section shows a more advanced use of the DRAM primitives to achieve a "virtual array" implementation. (A virtual array is an array of a size such that it would not ordinarily fit into a program's address space.) No new primitive operations are introduced in the virtual array example, but iterative remapping and other techniques extending the use of the primitives are illustrated. The example takes full advantage of the fact that a given virtual window, using as few as one APR, can be freely moved about within a large memory region.

Assume that a process needs 32K bytes of memory in order to create and manipulate a single large array and, consequently, would require four APRs if that much space were to be statically allocated to the program. Further assume that the array would not fit in the process's virtual address space if it were statically allocated (declared as a program variable), so dynamic RAM allocation and mapping are needed. The minimum requirement is that the static-process family to which the process in the example belongs does not exceed 28K words, leaving at least one unused APR available for window mapping.

Keeping the stated assumptions in mind, consider the following program segment:

```
[SYSTEM (MICROPOWER)] PROGRAM map6;

%INCLUDE 'MICROPOWER$LIB:DRAM.PAS'   { To get the "right stuff" }

TYPE
     huge_record = ARRAY [0..4095] OF INTEGER;

VAR
     i, off_set : INTEGER;
     q : ^ huge_record;
     rib1 : REGION_ID_BLOCK;
     window_base : UNSIGNED;

[INITIALIZE] PROCEDURE start;
BEGIN
          { Allocate 32 Kb of free RAM }
     IF NOT ALLOCATE_REGION (RIB := rib1, REG_SIZE := %O'1000')
          THEN ;
          { Map window to bottom of array initially }
     MAP_WINDOW (RIB := rib1, LENGTH := 8192,
                 OFFSET := 0, WINDOW_PTR := q);
     window_base := 0;
END;

          { Return Ith element as function value }
FUNCTION arry (index : UNSIGNED): INTEGER;
BEGIN
     IF NOT (((index - window_base) >= 0) AND
            ((index - window_base) < 4096))
          THEN
          BEGIN     { Not currently mapped, remap }
               UNMAP_WINDOW (WINDOW_PTR := q, LENGTH := 8192);
                 { Point to par tick }
               window_base := (INDEX DIV 32) * 32;
               MAP_WINDOW (RIB := rib1, LENGTH := 8192,
                           OFFSET := window_base DIV 32,
                           WINDOW_PTR := q);
          END;
     arry := q^[(index - window_base)]
END;
```

```
PROCEDURE set_arry (index,val : UNSIGNED);
BEGIN
     IF NOT (((index - window_base) >= 0) AND
           ((index - window_base) < 4096))
          THEN
          BEGIN     { Not currently mapped, remap }
               UNMAP_WINDOW (WINDOW_PTR := q, LENGTH := 8192);
                 { Point to par tick }
               window_base := (INDEX DIV 32) * 32;
               MAP_WINDOW (RIB := rib1, LENGTH := 8192,
                           OFFSET := window_base DIV 32,
                           WINDOW_PTR := q);
          END;
     q^[(index - window_base)] := val;
END;

BEGIN
          { Clobber the base to force a remap}
     window_base := 0;

          { Map, starting at arry[9984], the par tick just below
            arry[10000]}
     set_arry (10000,23);

          { No remap }
     set_arry (10002,25);

          { Still no remap, right at the end of the virtual address window}
     set_arry (14079,27);

          { Must remap, starting at arry[14080] }
     set_arry (14080,29);

          { No remap }
     set_arry (14081,31);

          { Clobber the base to force a remap }
     window_base := 0;

          { Map, starting at arry[9984], the par tick just below
            arry[10000]}
     IF arry(10000) = 23
          THEN  writeln ( 'success on 10000' )
          ELSE  writeln ( 'failure on 10000' ) ;

          { No remap }
     IF arry(10002) = 25
          THEN  writeln ( 'success on 10002' )
          ELSE  writeln ( 'failure on 10002' ) ;

          { Still no remap, right at the end of the virtual address window}
     IF arry(14079) = 27
          THEN  writeln ( 'success on 14095' )
          ELSE  writeln ( 'failure on 14095' ) ;

          { Must remap, starting at arry[14080] }
     IF arry(14080) = 29
          THEN  writeln ( 'success on 14096' )
          ELSE  writeln ( 'failure on 14096' ) ;
          { No remap }
     IF arry(14081) = 31
          THEN  writeln ( 'success on 14097' )
          ELSE  writeln ( 'failure on 14097' ) ;
END.
```

Chapter 6
Exception Processing

An exception is a significant event associated either with a processor trap (usually representing a hardware-detected error) or with a software-detected error or other special condition. The condition indicated by an exception, such as a memory fault or an arithmetic error, generally casts doubt on the ability of the running process to continue normal execution. Therefore, exceptions are essentially a kind of synchronous program interruption that, unlike I/O interrupts, cause a change in the flow of control within the currently running process. The kind of change depends on the kind of exception processing, if any, provided for the particular process and for the exception condition in question. The process may be switched to exception-wait state, to await "attention" by a separate exception-handling process; control may be redirected to the process's own exception service procedure; or the process may be aborted (forced to terminate abnormally into the inactive state). Note that any "unhandled" exception is fatal, causing the process to abort.

Despite the 16 types of exceptions defined for MicroPower/Pascal and the many possible exception conditions within a given exception type, all exceptions can be loosely grouped into two categories: hardware exceptions and software exceptions. (Software exceptions constitute by far the larger category.) The characteristics of each category are as follows:

- Hardware exceptions result from processor traps that cause an exception to be raised directly and unconditionally by the kernel. (All traps except IOT, debugger-set breakpoint, and power-fail/restart cause an exception, assuming that the corresponding trap handler has been included in the kernel.) Hardware exceptions represent either a hardware-detected error resulting from an instruction failure (an implicit error trap) or a special condition signaled by the intentional execution of a trap instruction in user code (an explicit "service trap"). Hardware-detected error conditions include bus timeouts, illegal or nonexistent addresses, illegal or reserved instructions, memory-parity errors, memory-management faults, and floating-point errors.

  The so-called service traps are caused by EMT, TRAP, and—possibly—BPT instructions. However, the BPT instruction is used by the PASDBG symbolic debugger for setting dynamic breakpoints and thus is not recommended for use in source code. (The remaining trap instruction, IOT, implements normal entry to the kernel for primitive services and cannot be used for any other purpose.) Such intentional traps can be used in a MACRO–11 program to trigger a user-defined service or other form of intervention from an exception-handling process. Although trap instructions cannot be coded directly in a Pascal program, they could be implemented if necessary with a MACRO–11 subroutine.

- A software exception represents an error or other special condition that is detected by software and is conditionally raised as an exception by means of the Report Exception primitive service (REXC$ macro request or Pascal REPORT procedure). In most cases, the condition is detected by a system software component, such as a kernel primitive or a system service process, which reports the condition back to the requesting process through a uniform status-return mechanism, leaving the exception reporting to the discretion of the user process.

  In principle, therefore, occurrence of a software-detected condition corresponding to a defined exception does not necessarily and automatically raise that exception. Except for kernel-detected stack violations, this principle holds across the board for a user process implemented in MACRO–11. For a Pascal-implemented process, however, many software exceptions (mostly relating to I/O and arithmetic functions) are unconditionally reported as such by the generated code. Some of the Pascal real-time service requests have an optional STATUS parameter that suppresses the otherwise implicit, automatic reporting of SYSTEM_SERVICE- and RESOURCE-type exceptions by the OTS interface routines that support the primitive service calls.

The kernel provides a mechanism for dispatching exceptions to either an exception-handling process or an exception service routine (Pascal exception procedure), as described in Section 6.3. If no form of exception handling is provided, the kernel aborts the faulting process, causing it to terminate in the inactive state.

The remainder of this chapter discusses exception types and codes, kernel exception dispatching, and exception handlers and service routines. Before reading further in this chapter, MACRO–11 programmers should be familiar with the CCND$, DEXC$, SERA$, and REXC$ primitive requests described in Chapter 3. Pascal programmers should be familiar with the corresponding requests (CONNECT_EXCEPTION, RELEASE_EXCEPTION, ESTABLISH, and REPORT) described in Chapter 17 of the *MicroPower/Pascal Language Guide*.

6.1 Exception Types and Codes

Exception conditions are identified by both a type and a code within the type. Of the 16 exception types, 12 are defined essentially for use by system software, 2 are reserved for future use, and 2 are dedicated to user-defined conditions. Each exception type is identified by an exception type symbol, such as EX$HIO in MACRO–11, and by a corresponding exception type symbol, such as HARD_IO in Pascal. The machine representation of a type symbol is a bit-mask word with a single bit set. Thus, type values can be used in MACRO–11 logical word instructions such as BIS and BIT. In equivalent Pascal terms, a type symbol represents a single element of a 16-member set, defined by the include file EXC.PAS as EXC_SET, and type values can be used in set expressions. (In some contexts, an exception type variable can validly represent several exception types, that is, the logical OR or union of two or more type symbols.)

Within each exception type, individual exception conditions are identified by an exception code. (Most of the 12 types defined for system use each comprise many specific exceptions.) The exception code symbols, of the form ES$xxx, are the same for both Pascal and MACRO–11 usage and represent unsigned word values. The internal format of an exception code value, composed of an encoded type portion and a subcode value, is described in Section 6.2. Normally, only the MACRO–11 programmer need be concerned with the composition of an exception code value.

Table 6–1 lists the exception types and codes and gives a brief description of each exception. The asterisks (*) in the table identify unconditional, kernel-raised exceptions that apply equally to MACRO–11 and Pascal processes. For MACRO–11 users, the exception type and code values are defined by the EXMSK$ macro in the COMU and COMM system macro libraries. For Pascal users, the corresponding exception types and codes are declared in the DIGITAL-supplied include file EXC.PAS. (EXC.DOC provides the commented form of the "condensed" EXC.PAS file.)

Note

> Do not confuse an exception type with an exception group, which is an attribute of a process. The exception group code is a value that is specified when a process is created or declared. The group code parameter (Pascal GROUP attribute) declares a process to be a member of a group for exception-handling purposes. (An exception group is a set of processes grouped together because of common exception-handling requirements.) The group code is then used to associate one or more exception handlers with a particular exception group. See Section 6.4.1 for further information.

Table 6–1: Exception Types and Codes

Type	Code	Description
EX$MEM*		MEMORY_FAULT
	ES$BUS*	Bus error: illegal address, bus timeout; trap to vector 4
	ES$MEM	Unspecified memory fault (subcode = 0); should never be encountered
	ES$MPT*	Memory parity error, where applicable; trap to vector 114
	ES$MMU*	Memory-management error, mapped targets only; trap to vector 250
	ES$VEC*	Vector fetch error, FALCONs only; trap to vector 0
EX$IOP*		ILLEGAL_OPERATION
	ES$FOP*	FP–11 or FPA floating-point opcode error; trap to vector 244
	ES$ILL*	Illegal or reserved instruction; trap to vector 10
	ES$IOP	Unspecified illegal operation (subcode = 0); should never be encountered
EX$EMT*		EMULATOR_TRAPMT instruction executed; trap to vector 30
	ES$EMT*	EMT instruction with a zero operand (subcode = 0)
	ES$xxx*	User-defined EMT exception codes, with a low-byte value from 1 to 255 matching EMT instruction operand
EX$TRP*		TRAP—TRAP instruction executed, trap to vector 34
	ES$TRP*	TRAP instruction with a zero operand (subcode = 0)

Table 6–1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$xxx*	User-defined TRAP exception codes, with a low-byte value from 1 to 255 matching TRAP instruction operand
EX$BPT*		BREAKPOINT_TRAP (not generated via PASDBG)
	ES$BPT*	User-coded BPT instruction executed; trap to vector 14
EX$HIO		HARD_IO—Hard I/O error conditions returned by a driver or communications process; corresponding exceptions are raised by the Pascal OTS
	ES$ABT	I/O request canceled by user or aborted by remote node
	ES$ATN	Device attention required
	ES$BOT	Beginning of tape
	ES$CTL	Controller error
	ES$DAL	Device already allocated
	ES$DRV	Drive error
	ES$EVL	End of volume
	ES$FOR	Format error
	ES$FRM	Framing error
	ES$HIO	Unspecified hard I/O error (subcode = 0); should never be encountered
	ES$IBN	Invalid block number
	ES$IDA	Invalid device address
	ES$IVD	Invalid data
	ES$IVM	Invalid mode
	ES$IVP	Invalid parameter
	ES$NXM	Nonexistent or read-only memory
	ES$NXU	Nonexistent unit
	ES$OFL	Device off line or not mounted
	ES$OVF	Data overflow
	ES$OVR	Device overrun
	ES$PAR	Parity error
	ES$PNA	Packet not available to support request
	ES$PWR	Device power failure

Table 6–1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$SPD	I/O processing stopped
	ES$TIM	Device timeout
	ES$UNS	Unsafe volume
	ES$WLK	Write-locked unit
EX$SIO		SOFT_IO—Soft I/O errors or special conditions, returned primarily by a driver, ACP, or communications process; the corresponding exceptions are raised by the Pascal OTS if the condition is unexpected. (Certain errors are detected as well as reported by the Pascal OTS, as noted.)
	ES$ABO	I/O aborted
	ES$BIV	Illegal Boolean value (Pascal only, detected by OTS)
	ES$DAS	Direct access requested on sequential file (Pascal only, detected by OTS)
	ES$DCF	Device full
	ES$DIO	Directory I/O error
	ES$DNU	Destination node is unreachable
	ES$DRF	Directory full
	ES$DVF	Attempt to signal device driver or ACP failed (detected by the OTS, ACP, or a communications process)
	ES$EOF	End of file encountered; not normally an error (detected by the ACP, conditionally reported by the OTS)
	ES$FAO	File already open (Pascal only, detected by OTS)
	ES$FIV	Illegal floating-point value (Pascal only, detected by OTS)
	ES$FNF	File not found
	ES$FNO	File not open (Pascal only, detected by OTS)
	ES$FNR	File not reset (Pascal only, detected by OTS)
	ES$FNW	File not rewritten (Pascal only, detected by OTS)
	ES$FRO	File is read-only: invalid write to OLD disk file (Pascal only, detected by OTS)
	ES$FVC	File-variable contention error (Pascal only, detected by OTS)
	ES$ICD	Invalid driver configuration data
	ES$IDR	Invalid directory format

Table 6–1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$IDS	Illegal device specification
	ES$IFN	Illegal function
	ES$IFS	Illegal file specification
	ES$IFW	Illegal field width (Pascal only, detected by OTS)
	ES$IIV	Illegal integer value (Pascal only, detected by OTS)
	ES$ILV	Illegal long integer value (Pascal only, detected by OTS)
	ES$INS	Invalid network specification
	ES$IRS	Illegal rename specification
	ES$IUP	Illegal use of UPDATE or DIRECT parameter (Pascal only, detected by OTS)
	ES$IVL	Invalid length specified
	ES$LRJ	Link rejected by remote task
	ES$NIP	No I/O in progress
	ES$NFS	Device not file structured
	ES$NRF	No reference data present
	ES$PAL	Path to remote task has been lost
	ES$PRO	File protection error
	ES$REF	Attempted read past EOF
	ES$RSZ	Record size of 0 specified (Pascal only, detected by OTS)
	ES$SIO	Unspecified soft I/O error (subcode = 0); should never be encountered
	ES$TNF	Task not found
	ES$UFN	Unsupported function
	ES$UIV	Illegal unsigned value (Pascal only, detected by OTS)
	ES$WEF	Attempted write past EOF
EX$NUM*		NUMERIC—Numeric errors reported either by the kernel (floating-point traps) or by Pascal runtime checks
	ES$CON*	Floating-point conversion error (FP–11 or FPA)
	ES$FDZ*	Floating-point divide by 0 (FP–11, FIS, or FPA)
	ES$FOV*	Floating-point overflow (FP–11, FIS, or FPA)
	ES$FUN*	Floating-point underflow (FP–11, FIS, or FPA)

Table 6-1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$IDZ	Integer divide by 0 (Pascal only)
	ES$INM	Modulus of negative integer (Pascal MOD function)
	ES$IOV	Integer overflow (Pascal only)
	ES$LDZ	Long integer divide by 0 (Pascal only)
	ES$LIC	Long integer to integer conversion error (Pascal only)
	ES$LNM	Modulus of negative long integer (Pascal MOD function)
	ES$LNP	Log of nonpositive value (Pascal LN function)
	ES$LOV	Long integer overflow (Pascal only)
	ES$LUC	Long integer to unsigned conversion error
	ES$NUM	Unspecified numeric error (subcode = 0); should never be encountered
	ES$SRN	Square root of negative value (Pascal SQRT function)
	ES$UDV*	Undefined floating-point variable (FP–11 or FPA)
	ES$UDZ	Unsigned divide by 0 (Pascal only)
	ES$UOV	Unsigned overflow (Pascal only)
EX$RSC		RESOURCE—Resource errors, either returned by a primitive or system process and optionally reported by the Pascal OTS, or detected and reported only by the Pascal OTS
	ES$DDP	DISPOSE of already disposed pointer (Pascal only)
	ES$LNR	Local node has no room for logical link
	ES$NFA	No free APR for window mapping
	ES$NFR	No free RAM available for a system process
	ES$NLZ	NEW request of length 0 (Pascal only)
	ES$NMB	Insufficient DATA_SPACE for I/O buffer (Pascal only)
	ES$NMC	Insufficient space for operation in RTACP pool
	ES$NMF	Insufficient DATA_SPACE for file variable (Pascal only)
	ES$NMK	Insufficient pool space for kernel structure
	ES$NMP	Insufficient DATA_SPACE for user structure (Pascal only)
	ES$NMS	Insufficient DATA_SPACE for process stack (Pascal only)
	ES$NNS	No network service process installed
	ES$RNR	Remote node has no room for logical link

Table 6–1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$RSC	Unspecified resource error (subcode = 0); should never be encountered
EX$RAN*		RANGE—Range errors, detected and reported by Pascal runtime checks only, except as noted
	ES$ASO	Array subscript out of bounds (Pascal INDEXCHECK option)
	ES$CSO	Case selector out of range (Pascal RANGECHECK option)
	ES$NIL	Reference of a NIL pointer (Pascal POINTERCHECK option)
	ES$PCC	Program consistency check; system-software error or version skew, should not occur
	ES$RAN	Unspecified range error (subcode = 0); should never be encountered
	ES$SEO	Set element out of range (Pascal RANGECHECK option)
	ES$STO*	Stack overflow (detected either by kernel or Pascal STACKCHECK option)
	ES$STU*	Stack underflow (detected either by kernel or Pascal STACKCHECK option)
	ES$VSE	Variable subrange exceeded (Pascal RANGECHECK option)
EX$EXC*		EXECUTION—Execution error, pertaining to a FALCON target configuration option only
	ES$BRK*	FALCON break trap, if configured; trap to vector 140
	ES$EXC	Unspecified execution error (subcode = 0); should never be encountered
EX$SVC		SYSTEM_SERVICE—System service errors returned mostly by primitive operations and optionally reported by the Pascal OTS; a few are detected and reported by the Pascal OTS only, as noted
	ES$AOV	Already owned vector, cannot connect
	ES$CDN	Cannot specify both descriptor and name (Pascal requests only)
	ES$EPN	Exception procedure not defined (Pascal REVERT request only)
	ES$IAD	Invalid address: odd or not in user address space
	ES$IPM	Illegal parameter
	ES$IPR	Illegal primitive
	ES$IST	Invalid structure descriptor
	ES$IVC	Illegal vector address
	ES$MDN	Must specify descriptor or name (Pascal requests only)

Table 6–1 (Cont.): Exception Types and Codes

Type	Code	Description
	ES$NID	No interrupt dispatch block (IDB) configured for vector; address not specified in DEVICES macro
	ES$RDE	Reply descriptor expected by RECEIVE_ACK (Pascal request only)
	ES$SIU	Structure is in use
	ES$SNI	Structure name already in use
	ES$SVC	Unspecified system service exception (subcode = 0); should never be encountered
EX$RS1		RESERVED_1—(Reserved by DIGITAL for future use)
EX$RS2		RESERVED_2—(Reserved by DIGITAL for future use)
EX$US1		USER_1—Type reserved for user-defined, user-reported exceptions
	ES$US1	Nonspecific exception of type USER_1 (subcode = 0)
	ES$xxx	User-definable USER_1 type exception code, with subcode value from 1 to 2047(decimal); see Section 6.2 for exception code format
EX$US2		USER_2—Type reserved for user-defined, user-reported exceptions
	ES$US2	Nonspecific exception of type USER_2 (subcode = 0)
	ES$xxx	User-definable USER_2 type exception code, with subcode value from 1 to 2047(decimal); see Section 6.2 for exception code format

6.2 Reporting Exceptions

Any process can deliberately raise an exception by issuing a Report Exception (REXC$) primitive request or the corresponding Pascal REPORT procedure. The REXC$ or REPORT parameters specify the exception type and code and, optionally, any additional arguments to be passed to an exception-handling process or exception service routine by means of the process stack. (A somewhat different but analogous interface exists in Pascal for passing additional arguments to an exception procedure.) See Chapter 3 of this manual or Chapter 17 of the *MicroPower/Pascal Language Guide* for detailed syntax of the REXC$ and REPORT calls, respectively.

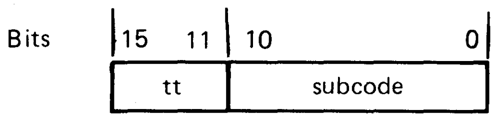
For a process implemented in Pascal, many OTS routines make implicit use of REPORT to raise software exceptions, either optionally or unconditionally as Table 6–1 indicates. (The STATUS parameter of many Pascal primitive requests may be used to inhibit the implicit reporting of RESOURCE and SYSTEM_SERVICE exceptions by the corresponding OTS interface routines.) No software exceptions other than kernel-detected stack violations are ever reported automatically for a process implemented in MACRO–11. When a software exception is raised by means of the REXC$/REPORT mechanism, the kernel dispatches the exception in the same manner as for a hardware exception.

As previously mentioned, an exception type and a code are both mandatory parameters to an REXC$ or REPORT call. In some circumstances, a process that chooses to raise an exception based on an exception code returned by a kernel primitive or a system process may need to derive an exception type value from the returned exception code before issuing the REXC$/REPORT call.

If the reporting process is implemented in MACRO–11, it may need to do so for any kind of error return—whether from a primitive or a system service process. (A MACRO–11 process receives only an exception code as the error return from either a primitive or a system process.)

If the reporting process is implemented in Pascal, it may need to do so only if it interacts directly with a device driver or other system process through the request/reply packet interface. (A Pascal process receives both an exception type and a code as the optional STATUS return from a primitive service call; the OTS interface routine performs the needed code-to-type translation.)

Every system-defined exception code value has type information encoded in a high-order field of the word value, permitting an exception-type mask value to be dynamically derived from the exception code. The format of an exception code value is as follows:



MLO–488–87

The high-order 5-bit field tt contains a value ranging from 1 to 16, corresponding to the bit to be set in the matching exception-type mask or to the matching exception-type set element in equivalent Pascal terminology. The subcode is a value that normally ranges between 1 and 2047 (3777 octal) and identifies a specific exception condition within the type. (Except for the codes ES$EMT, ES$TRP, and ES$BPT associated with trap instructions, an exception code with a subcode of 0 indicates an unspecified condition of a given type and is essentially a "placeholder" or origin value for meaningful codes of that type.)

Program fragments for dynamically deriving the matching exception type are shown in the next two subsections.

6.2.1 Deriving an Exception-Type Bit Mask in MACRO–11

The following program fragment assumes that R0 contains an exception code and that R1 and R2 are free for use. The exception code is retained in R1, and the derived exception-type mask is produced in R2, with R0 free for the subsequent REXC$ call. The fragment also assumes a non-EIS hardware environment—in particular, no ASH instruction.

```
        MOV     R0,R1         ; Save exception code intact

; Transform type field (highest 5 bits) in R0 into a rotate count
        SWAB    R0            ; Switch type field to the low byte
        BIC     #177407,R0    ; Isolate the encoded type value
        ASR     R0            ; Shift isolated value down 3 bits
        ASR     R0
        ASR     R0            ; R0 = encoded type in bits 0 to 4
```

```
; Construct type bit-mask in R2 corresponding to encoded type
        CLR     R2              ; Start off clean
        SEC                     ; Set C bit for initial rotate
RPTR:   ROL     R2              ; Form the bit mask
        SOB     R0,RPTR         ; Shift set bit as needed

; Report the exception condition
        rexc$s  mask=R2,code=R1,arglen=#0
```

6.2.2 Deriving an Exception-Type Set Value in Pascal

The following program fragment assumes that the variable except_code contains an exception code that might have been returned by a driver in an I/O-service reply packet. The derived exception-type set value is produced in the variable except_type, which may then be used in the subsequent REPORT call.

```
TYPE
        two_part = PACKED RECORD
                sub_cod   : [POS(0), BIT(11)] 0..2047;
                enc_typ   : [POS(11), BIT(5)] 1..16;
                END;
                                .
                                .
                                .
VAR
        except_type : EXC_SET;
        except_code : EXC_CODES;
                                .
                                .
                                .
BEGIN
                                .
                                .
                                .
except_type := [((except_code)::two_part).enc_typ-1)::EXCEPTIONS];

REPORT ( EXC_TYPE := except_type, EXC_CODE := except_code );
                                .
                                .
                                .
END.
```

6.3 Exception Dispatching

When an exception occurs, an exception stack frame is built on the running process's stack, and control passes to the kernel's exception-dispatching mechanism. The exception stack frame, described in Section 6.5, contains the exception type, exception code, and any additional arguments specific to the exception in question, as well as certain saved registers. The kernel first checks whether an exception-handling process is present for the group to which the running process belongs and for the type of exception to be handled. If such a handler exists, the kernel switches the running process (the process incurring the exception) from run state to the exception-wait-active state and places the process's PCB on the exception handler's exception queue semaphore, effectively signaling that semaphore and unblocking the exception handler.

The PCB contains the faulting process's latest context, of course, including its stack pointer (SP register). The kernel then declares a significant event, which causes the scheduler to switch a ready-active process (possibly the exception handler) into the run state.

At this point, the process causing the exception remains in exception-wait state, waiting for its exception to be processed and its eventual disposition to be determined. The exception handler, which was signaled by the kernel, is presumably in either ready-active or run state. Like any other process, the exception handler will run on the basis of its process priority relative to that of other processes on the kernel's ready-active queue.

If an exception handler does not exist for the exception group in question and for the type of exception to be handled, the kernel checks whether the running process has declared an exception service procedure. The kernel does that by examining the PCB of the running process. If the process has an exception procedure, established by the Set Exception Routine Address (SERA$) primitive or corresponding Pascal ESTABLISH procedure call, an exception service entry address will be present in field PC.EXC of the PCB. However, the kernel will not dispatch to that address unless the exception is of a type that is represented in the the PCB's exception-type bit mask (field PC.MSK). That bit mask reflects the exception type(s) specified in the SERA$ request or ESTABLISH procedure call that established the exception service routine or procedure.

When the exception procedure finishes its diagnostic or corrective processing, control returns to the point at which the exception occurred if the procedure ends normally—with an RTI instruction in the case of a MACRO–11 exception service routine. If the procedure issues a Stop Process request, however, control transfers directly to the process's termination entry point for a normal termination.

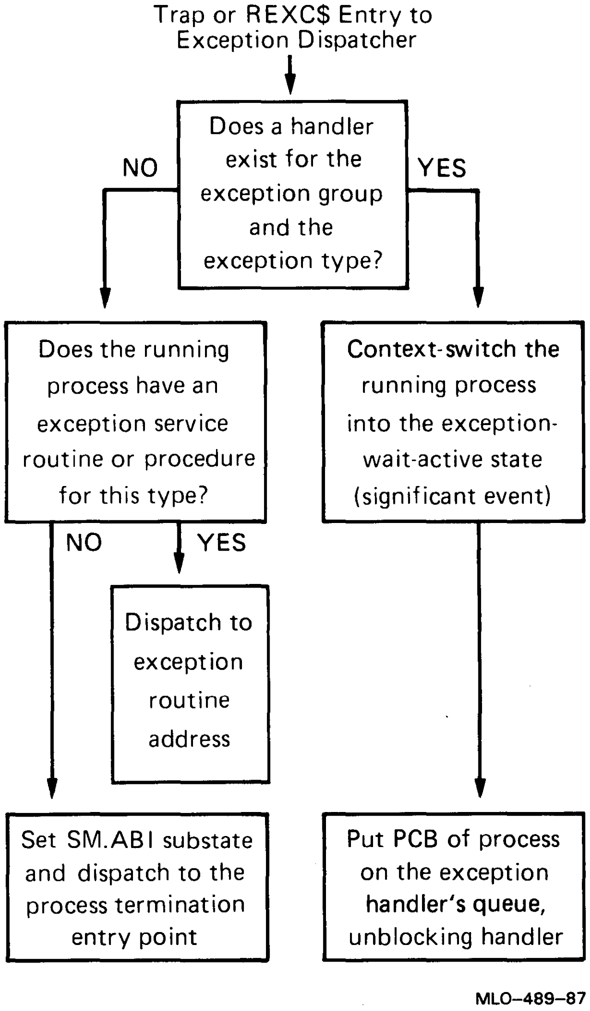
If neither an exception handler nor an exception routine/procedure is provided, the kernel immediately aborts the process by setting both the SM.ABO and SM.ABI state-code modifier bits in the process's PCB (field PC.STS) and directing control to the process's termination entry point (PC.TER) contained in the PCB. (The SM.ABI bit causes the process to terminate into the inactive state instead of simply disappearing when the Delete Process request is issued.) If you are debugging, PASDBG will report the fatal (unhandled) exception.

Note

> Unless the application was designed to allow the offending process to terminate without disrupting the application, continued execution of the application after a process has aborted will produce unpredictable results.

Figure 6–1 shows the action of the kernel in dispatching an exception.

Figure 6–1: Kernel Exception Dispatching


Trap or REXC$ Entry to Exception Dispatcher
Does a handler exist for the exception group and the exception type?
NO
YES
Does the running process have an exception service routine or procedure for this type?
Context-switch the running process into the exception-wait-active state (significant event)
NO
YES
Dispatch to exception routine address
Set SM.ABI substate and dispatch to the process termination entry point
Put PCB of process on the exception handler's queue, unblocking handler
MLO–489–87

6.4 Exception Handling

6.4.1 Exception-Handling Processes

An exception handler is a process that receives and manages specified types of exceptions for one or more groups of other processes. Note that in a mapped environment, an exception handler typically requires privileged (full system) mapping for direct access to PCBs.

An exception handler receives a pointer to the PCB of a faulting process on the queue semaphore that it has defined as its exception queue. After creating the required queue semaphore at start-up time, a process establishes itself or one of its subprocesses as an exception handler through use of the Connect to Exception Condition (CCND$) primitive request or the corresponding Pascal CONNECT_EXCEPTION procedure.

The CCND$ primitive associates an exception type and an exception group with a specified queue semaphore, thereby making that semaphore known to the kernel as the exception handler queue for that type and group. Several type/group combinations can be associated with the same exception queue, but any given type/group combination can have only one exception queue.

As described under exception dispatching, the kernel queues the PCB of a faulting process on the exception queue, if any, established for the type of exception that occurred and the group to which the faulting process belongs. (The kernel uses the standard signal-queue operation but puts a PCB pointer instead of a packet pointer on the queue.) The exception-handling process synchronizes with the arrival of exceptions through use of the Wait on Queue Semaphore (WAIQ$) primitive request or the Pascal GET_EXCEPTION procedure, an analog of GET_PACKET.

Every process has a group code in the range 1 to 255(decimal) associated with it when it is created. Essentially, group codes permit more than one exception handler for the same exception type to be present in the system, each implementing a different management strategy on behalf of the group(s) that it handles. The group code is defined for each process by the grp parameter of MACRO–11 DFSPC$ and CRPC$ requests or the GROUP attribute in Pascal PROGRAM headings and PROCESS declarations. The group parameter of a CCND$ or CONNECT_EXCEPTION request correspondingly specifies the group to be handled. Each Connect request can specify only one exception type and one group code. A number of Connect requests can be issued for a given queue semaphore, however, to establish the same exception queue for more than one group for an exception type and/or for more than one exception type for a group.

Alternatively, an exception handler can establish itself as the handler of a given exception type for all processes in an application, regardless of their group codes. That is done by specifying a group code of 0 (wildcard value) for the group parameter of the CCND$ or CONNECT_EXCEPTION request. Once a handler connects its exception queue to the wildcard group for a given exception type, no subsequent Connect requests for a specific group will be honored for the same exception type; conversely, no wildcard Connect request will be honored after a specific-group Connect request for a given exception type. That is to say, only one handler can be in effect for any combination of group and type, and the wildcard group code 0 implies all exception groups. (Group code 0 is invalid as a process attribute.)

Caution

> If a wildcard group handler itself causes an exception of a type that it handles, the handler process will lock up in a "fatal embrace" with itself and the process whose exception it was handling. (The handler will be blocked indefinitely on its own exception queue.) This situation can also occur if the handler is a member of any specific group that it handles, of course. Exception handlers in general should not cause exceptions in a debugged application.

When an exception occurs and an exception queue is signaled by the kernel, the exception handler waiting on that queue semaphore receives a pointer to the PCB of the process that caused the exception. That process has been blocked in the exception-wait-active state by the kernel.

The exception handler has access to the faulting process's exception stack frame through the user SP value saved in the PC.USV area (offset UC.SP) of the passed PCB, as well as access to all of the other saved context of the faulting process. (PCB offset symbols for Pascal coding are defined in the PCBM.PAS and PCBU.PAS include files for mapped and unmapped systems, respectively. See the corresponding PCBM.DOC or PCBU.DOC file for the commented form of those include files. The MACRO–11 offset symbols are defined by the PCBDF$ macro in the COMM and COMU macro libraries.)

Section 6.5 describes both the general format of an exception stack frame and specific stack frame formats for kernel-raised exceptions. In a mapped environment, the handler will already be mapped to the PCB but will need to map itself to the faulting process's stack area, using the stack pointer and mapping-context information in the PCB, assuming that the handler process is privileged and not a member of the same process family as the faulting process. Based on information in the stack frame and other information in or pointed to by the PCB, the handler must determine a course of action, "process the exception" in some fashion, and dispose of the PCB.

The exception handler can determine, by examination of PCB fields PC.EXC and PC.MSK, whether the process causing the exception is prepared to receive the exception for processing by its own exception service routine. The exception handler then must decide, based on that determination and the condition causing the exception, whether to pass the exception back to the process or to handle the exception by itself.

The exception handler uses the Dismiss Exception Condition (DEXC$) primitive request or the corresponding Pascal RELEASE_EXCEPTION procedure to dispose of the PCB and return the faulting process to the ready-active state for resumption of normal execution, further exception processing, or termination. The DEXC$/RELEASE_EXCEPTION options allow the exception handler three courses of action, as follows:

- Dismiss the exception (action = DISMISS). The process will resume execution at the point following the exception unless the saved user PC value was modified in the PCB by the exception handler.
- Pass the exception back to the process, causing it to resume execution at its exception service routine entry point (action = PASS). If the subject process has not declared an exception procedure that services the type of exception in question, this option has the same effect as ABORT, described next.
- Abort the process, causing it to resume execution at its termination entry point (action = ABORT). This action is appropriate if the exception represents a fatal situation that cannot be remedied by either the exception handler or the process causing the exception.

An exception handler must leave the exception stack frame on the process's stack exactly as received. The DISMISS and ABORT action options cause the necessary registers to be restored and the frame to be purged automatically. The PASS option leaves the stack frame for the process's exception routine or procedure if there is one. (If not, the PASS action is equivalent to ABORT.) In any case, the Dismiss Exception operation causes a significant event, returning the subject PCB to the kernel's ready-active queue for scheduling, unless the faulting process was suspended while in the exception-wait-active state.

The running priority of an exception handler should be higher than that of any process that may cause an exception that it handles.

6.4.2 Exception Service Routines and Procedures

Any process can establish an exception routine or procedure to service one or more types of exceptions caused by itself. A MACRO–11 process can have only one exception service routine, regardless of the number of exception types it wishes to manage internally. A Pascal process, however, can establish multiple exception procedures, up to the number of exception types to be serviced. As previously described, a process's own routine or procedure "receives" an exception condition either directly, if no external handler exists for the exception type and the process's exception group, or indirectly, if such a handler exists and exercises the PASS option for the exception in question.

6.4.2.1 Exception Routines in MACRO–11

An exception routine is essentially a synchronous "software interrupt" service routine that runs at process level on the process stack, has normal process mapping and several free registers (R3, R4, and R5), and can execute primitive operations if R0 is properly saved and restored. The routine is coded in assembly language as a closed subroutine that ends with a stack cleanup and an RTI instruction. (One exception routine code segment can be used by more than one process within a process family.)

A MACRO–11 process establishes its exception routine by issuing a Set Exception Routine Address (SERA$) primitive request, which specifies the routine's entry point and the type(s) of exceptions it will accept. This information is made available to the kernel by means of the PC.EXC and PC.MSK fields of the process's PCB.

When an exception is dispatched to an exception routine, whether directly or through an exception handler, the routine is entered in full process context, with SP pointing to the exception stack frame on the process stack. As shown in Figure 6–2, the stack frame contains the process's R3, R4, R5, PC, and PS context that was saved when the exception was raised, as well as the exception type, code, and variable-size argument list associated with the exception condition. General registers R3, R4, and R5 are therefore available for use by the routine without saving. If used, R0, R1, or R2 should be saved for restoring prior to exit unless the process is terminated by the exception routine. In particular, if any primitives are executed, R0 should be saved and restored around the primitive calls.

In its exception-specific code, the routine could perform some form of error logging and then force termination of the process with a Stop Process (STPC$S #0) request, which transfers control to the process's termination routine. Alternatively, the exception routine may take a corrective action where possible and appropriate to the particular exception condition. After corrective action is taken, the routine must do the following:

- Restore any registers it has explicitly saved
- Purge the stack down through the exception type, code, and argument list
- Restore and pop the saved R3, R4, and R5 values
- Exit by means of an RTI instruction, with SP pointing to the PC value

The RTI exit will return control to the process code at the point of the exception unless the exception routine has modified the saved PC value. (See the description of the ES$MMU exception stack frame in Section 6.5.3 for a case in which corrective action necessarily includes modification of the PC value.)

6.4.2.2 Exception Procedures in Pascal

For a process coded in Pascal, an OTS module implements the "outer shell" of the run-time exception routine mechanism described in the preceding section, concealing all hardware-level details from the Pascal programmer. (Register usage considerations and stack manipulations are all transparent, of course.) The OTS routine forms an interface between the kernel's exception dispatcher and one or more exception procedures explicitly declared and established as such in the Pascal program. The routine itself acts as a higher-level dispatcher, selectively calling a procedure based on exception type and executing the subsequent return, if any, to the process code. Functionally, then, the declared exception procedure(s) constitute the exception-specific "innards" of the run-time exception routine pointed to by field PC.EXC of the faulting process's PCB.

In the Pascal source code, the programmer declares one or more exception procedures for each process that will service its own exceptions. The procedures may be declared at outermost—main program—level for the static process or within a PROCESS declaration for a given dynamic process. As described in Chapter 17 of the *MicroPower/Pascal Language Guide*, such procedures have a formal parameter list consisting of a value parameter of type EXC_SET; a value parameter of type EXC_CODES; a value parameter of type UNSIGNED, representing an optional-argument byte count; and a VAR parameter of type UNIVERSAL, representing an optional-argument pointer variable. Further, an ESTABLISH procedure call is required for each declared exception procedure in order to establish each one as the procedure to be automatically invoked for one or more particular types of exceptions. Each ESTABLISH call specifies a procedure name and one or more exception type values, such as MEMORY_FAULT. The ESTABLISH call(s) must be executed within the process for which the corresponding exception procedure is to be established.

If an exception is dispatched to the faulting process, the procedure established for the corresponding exception type is invoked and is passed a set-element value indicating the exception type, an exception code, a byte count indicating the length of the optional arguments passed in the exception stack frame (may be 0), and a read-only pointer to the additional, exception-specific arguments, if any. The pointer value, of type UNIVERSAL, points to a variable of indeterminate type and size that is in fact the optional-argument portion of the exception stack frame, as shown in Figure 6–2. The pointer value will be undefined if the byte count is 0, that is, if no optional arguments have been passed in the stack frame, and the pointer must not be used in that case. (The UNIVERSAL type is predefined as [UNSAFE] INTEGER, which disables type checking for an entity of that type and effectively allows a formal parameter declared as such to accept and be used as a pointer of any type.) Thus, an exception procedure has access to all of the potentially useful information provided by an exception stack frame. If a given exception procedure is "interested" in interpreting optional-argument information, it can use the exception code and byte-count values as case selectors of an appropriate record type for the pointer.

A Pascal exception procedure is, however, generally more limited than a MACRO–11 exception routine in the range of actions it can take with respect to an exception representing an error, because of the difference in language level. Typically, an exception procedure is restricted to diagnostic action, some form of error logging, and process termination. If the procedure does not issue a STOP request, exit from the procedure will cause control to return either to the instruction following the one causing the trap, in the case of a hardware exception, or to the code following the REPORT_EXCEPTION operation, in the case of a software exception. In either case, and especially in the former, a return to "normal" processing following an error exception is usually undesirable.

If an exception procedure is implemented to handle exceptions resulting from Pascal I/O statements (for example, illegal integer value on a read), the exception procedure should not perform I/O. The indivisible nature of a Pascal I/O statement is inconsistent with the asynchronous characteristic of exception handlers.

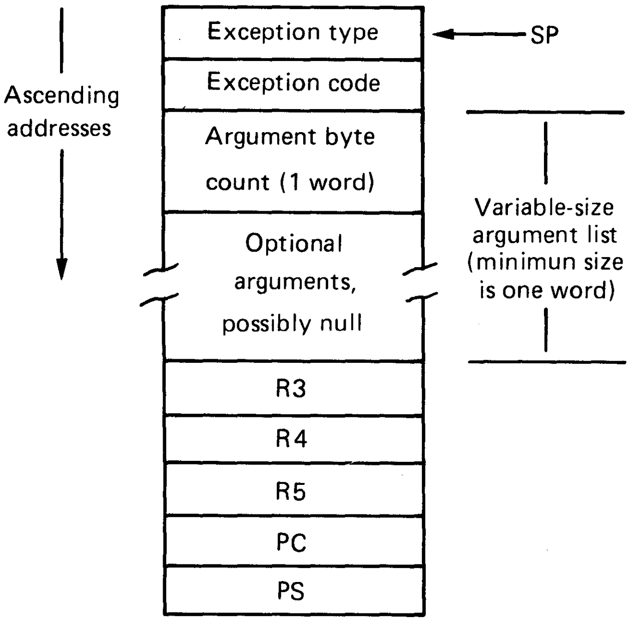
6.5 Exception Stack Frames

This section describes the general form of an exception stack frame and specific instances of the nonzero argument list contained in the frame for certain hardware exceptions.

6.5.1 General Stack Frame Format

Figure 6–2 shows the general form of the exception stack frame produced on the faulting process's stack by the kernel when an exception is raised.

Figure 6–2: Exception Stack Frame Format



MLO–490–87

The argument-list portion of the exception frame is variable in size, allowing the reporter of an exception condition to provide an arbitrary amount of information about it in addition to type and code. The minimum, or zero-argument, form of argument list consists of one word: an argument byte count of 0, indicating that the frame contains no optional arguments. The exception stack frame contains an argument list of that form for a large majority of system-defined exception codes.

6.5.2 Argument Lists for Software Exceptions

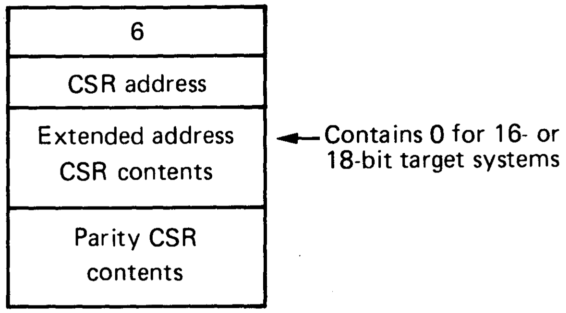
All software exceptions raised by system software components—the Pascal OTS and, in a very few cases, the kernel—have a null argument list. This category of exceptions includes all exceptions of types HARD_IO (EX$HIO), SOFT_IO (EX$SIO), RESOURCE (EX$RSC), RANGE (EX$RAN), and SYSTEM_SERVICE (EX$SVC) and exceptions of type NUMERIC (EX$NUM) other than floating-point errors.

The only software exceptions raised by the kernel are stack overflow (ES$STO) and stack underflow (ES$STU), of type RANGE, when such conditions are detected by the kernel through stack guardword checking during a context switch. Such checking determines only whether a stack boundary guardword has been overwritten, not whether the SP value has jumped around either guardword. (The same conditions may be detected and reported by optional, Pascal-generated range-checking code, which does not rely on the boundary guardwords.)

6.5.3 Argument Lists for Hardware Exceptions

The kernel produces optional arguments—an argument list with a nonzero argument byte count—for some of the hardware exceptions that it raises. The variable portion of the stack frame is described below for all hardware exceptions, grouped by exception type.

- MEMORY_FAULT (EX$MEM) Exceptions
  - ES$BUS, Bus errors—No optional arguments
  - ES$MPT, Memory parity errors on other than a KXJ11–CA:


6
CSR address
Extended address CSR contents
Contains 0 for 16- or 18-bit target systems
Parity CSR contents

MLO–491–87

As indicated in the diagram, the value of the second argument is significant only for 22-bit target systems.

For the KXJ11–CA, there are no optional arguments.

- ES$MMU, Memory-management unit errors:

8
MMU status register 3
MMU status register 2
MMU status register 1
MMU status register 0

MLO–492–87

The PC value saved on the stack for an ES$MMU exception does not point to the beginning of the instruction that caused the protection fault but rather to an intermediate point within the instruction. However, the content of MMU status register 2 is the proper instruction address. Therefore, if an ES$MMU exception handler or service routine takes corrective action in order to return control to the offending instruction, the handler or routine must, prior to exit, replace the saved PC value with the value of the second argument provided in the argument list.

See Section 6.5.4 for a description of special-case ES$MMU exception processing by the kernel involving a corrupted or marginal user SP value.

- ES$VEC, Vector fetch error (FALCON or FALCON–PLUS only)—No optional arguments

- ILLEGAL_OPERATION (EX$IOP) Exceptions
  - ES$ILL, Illegal instruction errors—No optional arguments
  - ES$FOP, Floating-point opcode error—No optional arguments
- EMULATOR_TRAP (EX$EMT) Exceptions
  - ES$EMT, Zero-operand EMT instruction executed—No optional arguments
  - ES$xxx, EMT instruction executed with the nonzero operand xxx—No optional arguments
- TRAP (EX$TRP) Exceptions
  - ES$TRP, Zero-operand TRP instruction executed—No optional arguments
  - ES$xxx, TRP instruction executed with the nonzero operand value xxx—No optional arguments

- BREAKPOINT_TRAP (EX$BPT) Exceptions
  - ES$BPT, BPT instruction executed—No optional arguments
- NUMERIC (EX$NUM) Exceptions
  - Floating-point processor faults for conversion errors (ES$CON), overflow (ES$FOV), underflow (ES$FUN), divide by 0 (ES$FDZ), and undefined variable (ES$UDV):

2
Address of faulty instruction

MLO–493–87

  (All other exceptions of type NUMERIC represent software-detected errors.)

- RANGE (EX$RAN) Exceptions
  - ES$STO, Process stack overflow—No optional arguments
  - ES$STU, Process stack underflow—No optional arguments

  (All other exceptions of type RANGE represent software-detected errors.)

- EXECUTION (EX$EXC) Exceptions
  - ES$BRK, Break trap (if configured, on a FALCON or FALCON–PLUS only)—No optional arguments

6.5.4 Special Cases of MMU–Fault Exception Processing

To prevent a fatal trap-to-4 or MMU trap from occurring within the kernel's exception dispatcher as a result of nested traps, the kernel takes special action in a mapped application for the two cases described below. In both cases, the faulting process's stack is invalidated, and no run-time recovery is possible:

- If an ES$MMU exception occurs—presumably as a result of a corrupted user stack pointer—and the kernel detects that the user's SP value is obviously invalid, that is, not within the user's address space, the kernel resets the user SP to the initial beginning-of-stack value before pushing the exception stack frame. Thus, the MMU register-2 value will point to the offending instruction in the user's address space, but the user's stack will contain nothing below the MMU exception stack frame.
- If any type of exception occurs and the user SP value points close enough to the process's address-space limit to cause an MMU trap while the kernel is pushing the exception stack frame for the initial exception, the kernel ignores the initial exception, resets the user SP to the beginning-of-stack value, and processes the MMU trap as if caused by the faulting process. Thus, an ES$MMU exception is raised against the process causing the initial exception, but the PC and the MMU register values in the exception stack frame will point into kernel address space, and the user's stack will contain nothing below the MMU exception stack frame.

Chapter 7
Interrupt Dispatching and Interrupt Service Routines

This chapter discusses MicroPower/Pascal device driving, focusing on interrupt service routines (ISRs) associated with device drivers. Before reading this chapter, you should read the descriptions of the CINT$ and DINT$ kernel primitives and the FORK$ and P7SYS$ kernel services in Chapter 3. See also the description of driver/ISR mapping in Chapter 2.

7.1 Device Interrupts and Device Driver Functions

An interrupt is essentially a processor trap caused by a request for service or attention generated by an external device. Depending on the device's hardware priority and the priority at which the processor is operating, the interrupt request is serviced either immediately, if it is from the highest-priority device requesting the processor, or later, following the current higher-priority operation. Interrupts serviced in that manner permit prioritized, event-driven processing involving I/O devices.

An interrupt is serviced by the processor after completion of the current instruction—excepting floating-point instructions that can be interrupted during their execution. Once the instruction execution has 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 originally interrupted program is restored, and processing continues as before.

I/O device handling in MicroPower/Pascal applications is done by a device driver—a static process that includes an interrupt service routine (ISR) and an optional fork routine. The ISR is an essential part of the real-time environment of MicroPower/Pascal applications. The fork routine, an extension of the ISR, permits critical processing to be done within the ISR itself, followed by less critical processing in the fork routine. Fork routines can be interrupted by any interrupt request. Thus, interrupt latency—the time between a hardware-generated interrupt request and ISR entry—can be kept to a minimum, an important requirement in many real-time applications.

An interrupt results in entering an ISR only when an interrupt by the device is expected by a given device driver. The Connect to Interrupt (CINT$) kernel primitive permits a driver to connect its ISR to a particular interrupt vector. Interrupt requests for the device configured to that vector result in service by the ISR only after the CINT$ primitive has been issued. If an

ISR has not been connected to an interrupt vector or has been disconnected by the DINT$ primitive, the interrupt is vectored to the default null ISR, which dismisses the interrupt without performing any useful processing.

Entry to the driver's ISR or to the null ISR is made by means of an interrupt vector in low memory. One or more vectors for each hardware device are defined for the application by editing the system configuration file. This file must be edited to reflect the device hardware configuration before building the kernel and driver software.

ISRs for device drivers written in Pascal must be written in MACRO-11 and merged with the program as an external procedure. The MicroPower/Pascal CONNECT_INTERRUPT procedure call effectively ties the "procedure" in with the rest of the Pascal driver program. Typically, the MACRO-11 ISR will be written in position-independent (PIC) code so the build-time constraints of relocating the ISR code and data to the PAR 2 and 3 address ranges, respectively, are avoided. (The build-time relocation requirements can be very difficult to satisfy in the case of a Pascal-implemented driver program.) The kernel-mode mapping of ISR code and data into PAR 2 (code) and PAR 3 (data) values in a mapped environment is described in Chapter 2. When an ISR is coded as PIC, the CONNECT_INTERRUPT procedure call or corresponding CINT$ primitive request dynamically relocates the ISR code and data appropriately on interrupt, using the contents of the corresponding user-mode PARs for the virtual-address transformations, as described for the CINT$ primitive in Chapter 3.

ISRs for device drivers written in MACRO-11 can be written as either PIC or non-PIC. In the case of an ISR written in PIC, the CINT$ primitive translates the virtual addresses as described for a device driver written in Pascal.

7.2 Interrupt Service Routine

The ISR has the following functions:

- To respond quickly to interrupt requests generated 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 interrupt servicing code
- To perform, at interrupt level, some critical time-dependent processing
- To perform, at fork level, the remaining part of the time-dependent processing required to completely service the interrupt and to signal the driver process that the event (interrupt) has occurred

In essence, the ISR and associated fork routine processes hardware events as they occur in real time. Less critical I/O processing, such as replying to another process's I/O request, is done by the device driver at process level.

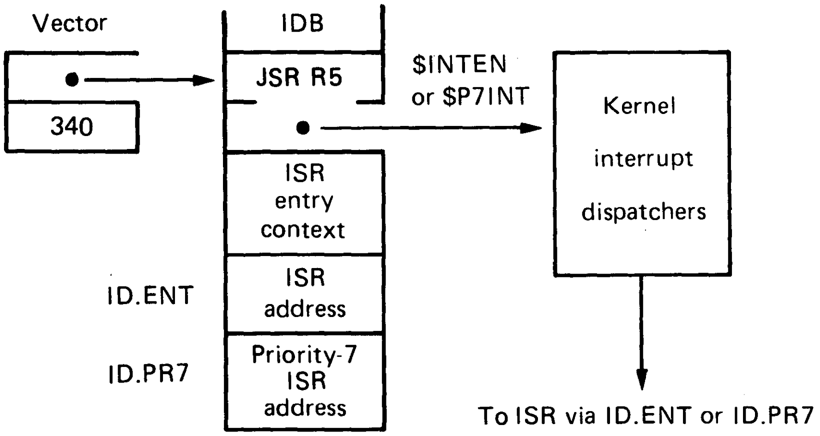
The proportion of I/O processing performed by the ISR and by the driver process is determined largely by the characteristics of the device serviced and the nature of the I/O transfer. In general, deferring as much I/O processing as possible to process level as opposed to interrupt or fork level (in the 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 a transfer. As a result, the only interrupt-level processing that need be done is to awaken the driver process when a transfer completes. The ISR issues a FORK$ request—enters fork-level processing—and signals the driver process, indicating that an interrupt has occurred, that the requested I/O transfer has completed, and that I/O processing at process level can continue.

In contrast, a device that transmits only one or two bytes for an interrupt at a relatively high and fixed transfer rate may require that almost all of its I/O processing be done at interrupt level. This requirement is based on the excessive context-switching time that would otherwise be required by the frequent interrupts if data were processed by the device driver at process level, even assuming that the system load would allow the process to keep up with the interrupts. Thus, once a read or write operation is initiated by the driver process, the ISR iteratively handles the transfers to or from a device and a buffer and signals, at fork level, the driver process when the operation is completed. In addition, the ISR informs the driver process that the transfer was either successful or that an error was detected. In case of an error, the process-level I/O handling performed by the driver typically is responsible for determining whether a retry is required and, if so, for initiating the retry operation.

7.3 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 by means of the interrupt dispatch block (IDB) and the kernel interrupt dispatcher is as follows:



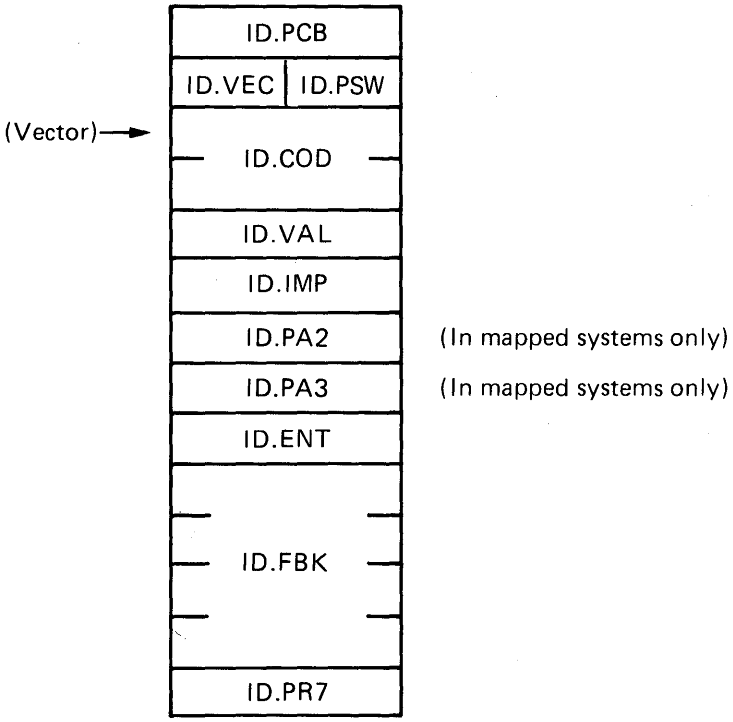
MLO–494–87

7.3.1 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 user-determined 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.

The format of an IDB is:



MLO–495–87

In the previous format:

- ID.PCB is the pointer to the PCB of the process owning the vector, if an owner currently exists—that is, the process that performed the connect-interrupt operation.
- ID.PSW is the desired value of the PSW priority and condition code (CC) bits on entry into the ISR.
- ID.VEC is the scaled address of the vector associated with this IDB (address/2)—effectively, a backpointer to the corresponding vector.
- ID.COD is the JSR R5,@#$INTEN (or JSR R5,@#$P7INT) instruction (two words).
- ID.VAL is the value to be passed in R4 on normal ISR entry; the value is specified in the CINT$ call.
- ID.IMP is the pointer to the impure area; the location is specified in the CINT$ 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 a normal ISR's entry point (that is, with priority less than 7).

- ID.FBK is the start of the fork block for this interrupt (four words).
- ID.PR7 is the address of a priority-7 ISR's entry point.

7.3.2 Kernel Interrupt Dispatcher

An interrupt at a given vector causes processor control to pass to field ID.COD in that vector's IDB. The IDB field contains either a JSR R5,@#$INTEN or a JSR R5,@#$P7INT, causing a transfer to the kernel's interrupt dispatcher, with R5 pointing to field ID.VAL in the IDB.

The interrupt dispatcher has two entry points: the normal entry point ($INTEN) for dispatching to ISRs that execute at a CPU priority less than 7 and a special entry point ($P7INT) for dispatching to ISRs that initially execute at priority 7. In either case, the dispatcher is entered with interrupts inhibited at CPU priority 7. The correct entry point for a particular interrupt is set in the corresponding IDB.

When entered for a normal ISR, the interrupt dispatcher saves the full ISR hardware context (R3, R4, R5, and, if mapped, 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 APRs 2 and 3 in the kernel-mode mapping.) The interrupt dispatcher 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 context-switching procedure—except for 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 a priority-7 execution, no other interrupts can be serviced until ISR execution is completed; thus, much care must be taken to keep the ISR code that is executed at priority 7 as brief as possible.

To summarize, the interrupt dispatcher performs the following functions:

- When dispatching to normal ISRs, saves on the stack general registers R3 and R4 and, in mapped systems, kernel-mode mapping registers PAR2 and PAR3; R5 is automatically saved by the JSR R5 instruction in the IDB.

  When dispatching for priority-7 interrupts, saves general register R4 as well as kernel-mode PAR2 and PAR3 in mapped systems; R5 is automatically saved by the JSR R5 instruction in the IDB.
- Increments the interrupt nesting level. The interrupt level is initialized to –1 by INIT, which represents process level (no interrupts or primitive requests being serviced). Assuming that user code is currently executing, the first interrupt raises the nest level to 0, which marks the transition from process level to either kernel-primitive or interrupt level.
- In an unmapped system, switches to the system interrupt stack if the nesting level is incremented to 0. In both mapped and unmapped systems, if the nesting level is incremented to 1 and a kernel-primitive execution was interrupted, the stack is switched from the per-process kernel stack in the process's PCB to the system interrupt stack.
- In a mapped system, establishes the mapping context of the ISR: kernel-mode PARs 2 and 3.
- 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.

- For a normal ISR, dispatches to the ISR, with R3 pointing to the impure area, R4 containing the value passed in the CINT$ 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. Any other registers must be saved before use and restored afterwards.

  For a priority-7 ISR, dispatches 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, excepting APR 2 and 3 in a mapped system. R5 is available for use without explicit saving and restoring, provided that the ISR is not going to issue a P7SYS$ followed by a FORK$. All other registers must be saved before use and restored afterwards.

7.3.3 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 aids your overall comprehension of MicroPower/Pascal interrupt handling and the functions of the CINT$ primitive.

7.3.3.1 Allocating IDBs and Setting Vectors

Every interrupt vector that will be used in an application must be declared in the system configuration file so a corresponding number of IDBs can be allocated during application building. Those vectors are declared in the DEVICES macro. Each declared vector points to a unique IDB when the kernel memory image is constructed. (In most 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.)

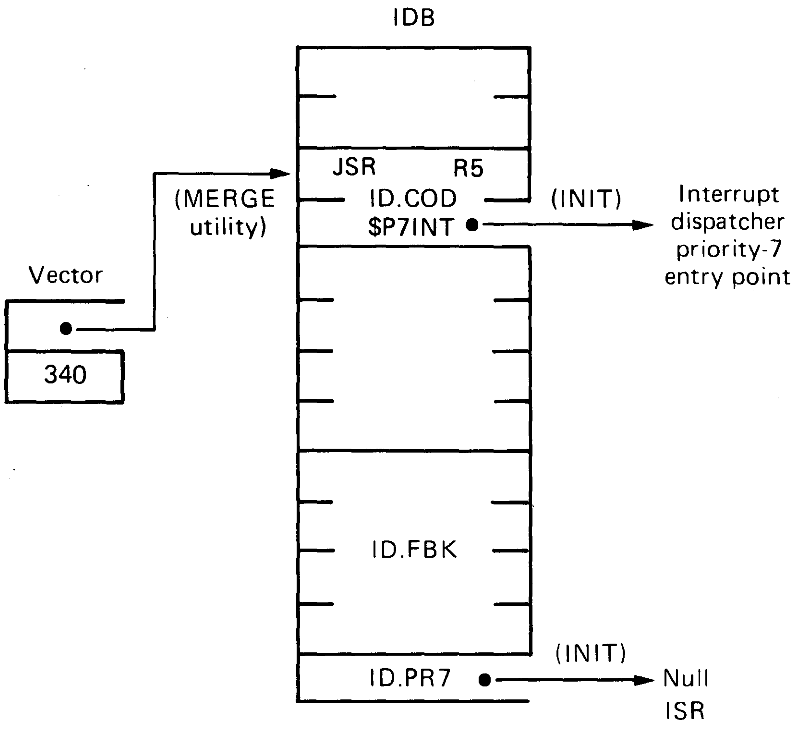
All vectors not declared as part of the hardware configuration point to a single null IDB; any undeclared interrupts will dispatch to the null ISR. The null ISR, located in the kernel, runs at priority 7, increments a counter, and returns—executes an RTS R4 instruction—thereby dismissing the interrupt.

Since IDBs are dynamically modified by the CINT$ kernel service, IDBs must always be in RAM.

7.3.3.2 Initializing IDBs During Start-Up

IDBs are statically allocated in the kernel's impure-data segment but are not statically initialized; that is, IDB content is undefined at build time. The system-initialization routine, INIT, initializes all IDBs during the start-up/restart sequence by directing each IDB to the null ISR.

INIT places a JSR R5,@#$P7INT instruction in IDB field ID.COD, which passes control to the interrupt dispatcher's priority-7 entry point when an interrupt occurs. INIT then places the null ISR's entry address in IDB field ID.PR7, which is used for dispatching priority-7 ISRs. Thus, after system start-up, any unsolicited interrupts from declared vectors—as well as unexpected interrupts from undeclared vectors—are ignored until a proper connection has been made between a particular interrupt vector and an ISR, by means of the CINT$ primitive. The linkages in place after system initialization are as follows:



MLO-495-87

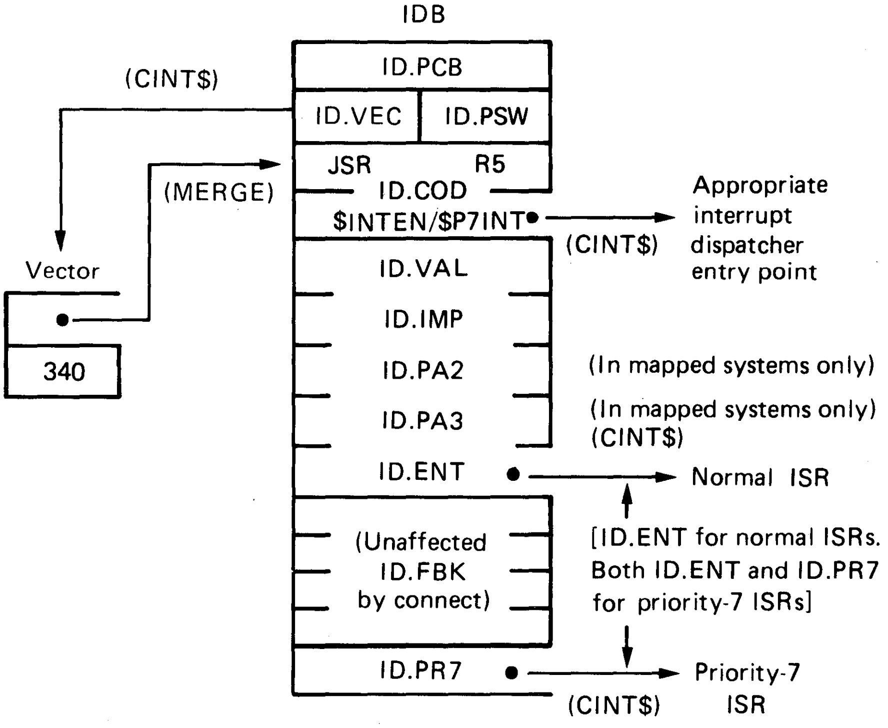
7.3.3.3 Connecting Interrupts to ISRs

Each IDB remains in the state shown previously, 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 with the implied ISR in the case of CONNECT_SEMAPHORE, by modifying the IDB assigned to the vector.

Specifically, executing the CINT$ primitive does the following:

- 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 (less than or equal to 7, respectively).
- 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.
- Links the IDB back to its vector through the ID.VEC field.
- Sets all other fields of the IDB, except ID.FBK, to the values specified or implied in the CINT$ call; those fields contain ISR context-related information. (The fork-block substructure is not involved in the set-up operation.)

The result of connecting the interrupt to the ISR is:



MLO–497–87

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.

7.4 Entering and Executing ISRs

7.4.1 Entering and Executing Normal ISRs

The code fragment in Example 7–1 shows the typical form of an ISR. 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 contained 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.

Kernel primitives cannot be called from ISRs at interrupt level. To prevent kernel reentrancy problems, an ISR must issue a FORK$ service call before issuing any kernel primitive. That guarantees that an ISR issuing a primitive request will not usurp a process-level primitive operation, causing the kernel to be invalidly reentered. However, interrupts can be nested; that is, higher-priority interrupts can be serviced without causing any reentrancy problems in the kernel.

Note

> In a normal ISR, R3, R4, and R5 can be used without first saving their content. However, those registers may contain information needed by the ISR and, thus, may require saving anyway. In a priority-7 ISR, only R5 can be used without first saving its contents. However, the contents of R5 must be saved and restored before using a P7SYS$ call.

7.4.2 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 full interrupt entry procedure. ISRs servicing 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 and should use as little time as possible at priority 7.

When the interrupt dispatcher enters 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 to 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 P7SYS$ kernel service can be called from priority-7 ISRs while executing at priority 7, since they do not have sufficient context. Thus, to issue a fork request, which must precede any primitive request, the ISR must first issue a P7SYS$ call. The P7SYS$ 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.

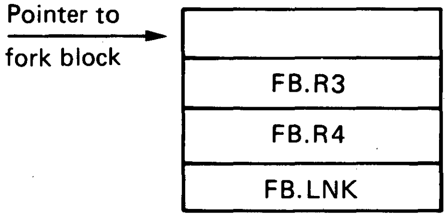
7.4.3 Fork Routine

An ISR must issue a fork request before requesting any other kernel service, that is, any primitive service. 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 CPU priority 0, although still in kernel mode. The fork request is made by issuing a $FORK call, with R5 pointing to the fork block, as it does on entry to a normal ISR. The stack must be purged of any data pushed on it by the ISR before issuing the FORK$ call.

Chapter 3 describes the FORK$ primitive. Appendix A describes the overall scheduling hierarchy.

The kernel responds to the FORK$ call by queuing the request on the kernel's fork-request queue. The context of the ISR (R3, R4, and PC) is preserved in the fork block, including the ISR's kernel-mode mapping in the case of a mapped system.

The format of a fork block is as follows:



MLO–498–87

In the previous format:

- 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.

Fork routines must not issue kernel primitives that may cause them to block. Fork routines may perform Signal operations, for example, on semaphores but may not wait on a semaphore, which could block if no signal were pending. Conditional waits may, however, be used, since they never block.

The fork-queue overrun mechanism, described under the FORK$ request in Chapter 3, provides a kind of synchronizing function for ISRs that service a device with a high interrupt rate, such as a fast serial or parallel line that signals the receipt of every byte or word. The fork overrun essentially indicates that a previously issued fork request is still in the queue. (One fork routine can also be in progress and have been interrupted.) The occurrence of an overrun—a C-bit return from the FORK$ request—allows the ISR code to store the data just received and to increment a counter—for example, for the pending fork routine—assuming that the fork routine has been designed to operate in an iterative fashion. Due care must be taken in the instructions used to implement the coordination in both the ISR and fork routine coding, to prevent a race condition when a dequeued fork routine has been interrupted in progress.

The kernel's clock service routine, shown in Example 7–1, illustrates the interaction between ISR and fork-level code with respect to fork overruns. An asterisk in the comments indicates a line of code executed at CPU priority 7, and an asterisk in parentheses (*) indicates a line of code executed at a CPU priority less than 7 but greater than 0. Thus, all lines of code having an asterisk in the commentary constitute ISR coding. The remaining code forms the fork routine.

The decrementing of the timeout value—or expiration count—$EXPCL and $EXPCH by the ISR and the incrementing and testing of the same value by the fork code implements the coordination between the two code segments in a simple, raceproof, but subtle manner.

Example 7-1: Kernel Clock ISR and Fork Routine

```
        .enabl     GBL
        .mcall MACDF$,P7SYS$,FORK$,SGNL$S,IDBDF$,MISDF$,TIMDF$
        macdf$
        idbdf$
        misdf$
        timdf$
        .globl     $TIME
        orig$    .20CL1,<R0,GBL,I>
;+
; Module name: CLK60.MAC
;
; System: MicroPower/Pascal
;
; Functional Description:
;
;       This module contains the clock interrupt-service-routine front end
;       for 60 Hertz systems.  It keeps the system time in milliseconds.
;       To do this, it must add 17, 17, 16 for each group of 3 ticks.
;       This is because ( 17+17+16 ) * ( 60/3 ) = 1000 milliseconds
;       after 60 ticks.
;-

        .sbttl     Declarations
;+
; Local macro definitions - None
;-

;+
; Data owned by this module is defined here and storage is allocated in
; the appropriate data section.
;-
        dat$

TICCNT: .word 0  ; Tick counter

        .sbttl    Initialization code
;+
; Code to initialize data defined in this module.  (This code constitutes
; a portion of the kernel INIT routine and performs a sufficient subset of
; the CINT$ primitive function.  Note that all IDBs must have already been
; set up by preceding INIT code.)
;-

        ini$ ,5

$OCIDB:

; Initialize clock IDB

        MOV @#V.CLK,R0                  ; R0 -> IDB for clock
        MOV #$CLK60,ID.PR7-ID.COD(R0)   ; Dispatch to system clock
                                        ;  ISR on interrupt
        MOV #3,TICCNT                   ; Initialize tick counter

        psect$ *
```

(Continued on next page)

Example 7-1 (Cont.): Kernel Clock ISR and Fork Routine

```
        .sbttl    Clock ISR Front End
$CLK60::
        DEC TICCNT                     ;* Add 16 milliseconds this tick?
        BNE 20$                        ;* If NE, no
        MOV #3,TICCNT                  ;* Reset tick counter

        assum$ TM.SIZ EQ 6

        ADD #16.,$TIME+TM.LOW          ;* Add 16 millisecs to low-order time
        ADC $TIME+TM.MID               ;* Add carry to middle-order time
        ADC $TIME+TM.HIG               ;* Add carry to high-order time
        SUB #16.,$EXPCL                ;* Decrement low-order timeout value
        SBC $EXPCH                     ;* Decrement high-order timeout value
        BMI $CLKFK                     ;* If MI, must fork to check for
                                       ;*   timeout expirations
        BNE 10$                        ;* If NE, no expiration yet
        TST $EXPCL                     ;* Exact expiration?
        BEQ $CLKFK                     ;* If EQ, yes
10$:    RTS R4                         ;* Dismiss priority-7 interrupt

20$:    assum$ TM.SIZ EQ 6

        ADD #17.,$TIME+TM.LOW          ;* Add 17 millisecs to low-order time
        ADC $TIME+TM.MID               ;* Add carry to middle-order time
        ADC $TIME+TM.HIG               ;* Add carry to high-order time
        SUB #17.,$EXPCL                ;* Decrement low-order timeout value
        SBC $EXPCH                     ;* Decrement high-order timeout value
        BMI $CLKFK                     ;* If MI, must fork to check for
                                       ;*   timeout expirations

        BNE 30$                        ;* If NE, no expiration yet
        TST $EXPCL                     ;* Exact expiration?
        BEQ $CLKFK                     ;* If EQ, yes
30$:    RTS R4                         ;* Dismiss priority-7 interrupt

$CLKFK:                                ;* Common clock fork routine is con-
                                       ;* tained in psect .20CL2 in source
                                       ;* module CLKISR.MAC.

        orig$ .20CL2,<RO,GBL,I>
;+
; Module name: CLKISR.MAC
;
; System: MicroPower/Pascal
;
; Functional Description:
;
;       This module contains the common clock-interrupt-service code and
;       fork routine for all clock frequencies.  It keeps the system time
;       in milliseconds.  It must be merged with a frequency-specific front
;       end.  P-sect ordering is crucial; the clock front ends are contained
;       in section .20CL1 and this module is in section .20CL2.
;-
```

(Continued on next page)

Example 7–1 (Cont.): Kernel Clock ISR and Fork Routine

```
        .sbttl Common Clock ISR

$CLOCK::
        P7SYS$ 6                       ;* Drop priority to do fork
        FORK$                          ;(*) Fork
        BCS 70$                        ;(*) If CS, then reentrant fork rtn.
                                       ;  will wake up the next process too
        MOV $SYSTM,R4                  ; R4 -> Head of timer blocking queue
        CLR R3                         ; R3 = Number of signals to do
        MOV @R4,R4                     ; Any sleeping processes?
        BEQ 60$                        ; If EQ, no
        BR 20$                         ; Merge in loop

10$:    ADD PC.TML(R4),$EXPCL          ; Expired?
        ADC $EXPCH                     ;
        ADD PC.TMH(R4),$EXPCH          ; Expired?
20$:    TST $EXPCH                     ; Expired?
        BMI 25$                        ; If MI, yes
        BNE 30$                        ; If NE, no
        TST $EXPCL                     ; Expired?
        BNE 30$                        ; If NE, no
25$:    INC R3                         ; Yes, do a signal
        MOV @R4,R4                     ; Any sleeping processes?
        BNE 10$                        ; If NE, yes
30$:    MOV $SYSTM,R4                  ; R4 -> Timer semaphore
        TST R3                         ; Wake anybody up?
        BEQ 50$                        ; If EQ, no
40$:    CALL $SGNLS                    ; Call stack form of SGNL$ primitive

;+
;Following is what you would do if from a driver ISR fork routine. Use the
;call $SGNLS instead, since we are in the kernel.
;       SGNL$S #$SYSTM                 ; Wake up process
;       BCS
;+

        SOB R3,40$                     ; Wake up R3 # of processes
50$:    TST @R4                        ; Anybody left on the queue?
        BNE 70$                        ; If NE, yes
60$:    MOV #77777,$EXPCH              ; No, don't wake me up for a few days
70$:    RETURN

        .end
```

7.4.4 Dismissing an Interrupt

The normal ISR dismisses an interrupt by issuing an RTS PC instruction. Before dismissing the interrupt, however, the routine must first clean the stack and restore any registers that were saved by the ISR on entry. The interrupt can be dismissed in this manner either at interrupt level or when executing a fork routine.

The priority-7 ISR dismisses the interrupt by issuing an RTS R4 instruction. Before dismissing the interrupt, however, the routine must also clean the stack and restore any registers that were saved by the ISR on entry.

7.5 Kernel Interrupt Exit Processing

A return from an ISR is always made by means of either an RTS PC instruction (normal ISRs) or an RTS R4 instruction (priority-7 interrupts). If a priority-7 ISR issues the P7SYS$ call, it becomes a normal ISR, exiting by way of 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:

- If the return is to an interrupted ISR, the kernel decrements the interrupt nesting level and restores R3, R4, and R5, as well as kernel mapping registers APR 2 and APR 3 in mapped systems.
- If the return is 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 APR 2 and APR 3 in mapped systems.
- If the return is to a fork routine, the kernel decrements the interrupt nesting level and resumes the interrupted fork routine execution. If one or more additional fork routines are queued, the interrupted fork routine is executed first, followed immediately by the remaining fork routines, according to their order of placement on the fork queue. No return is made to process-level execution until all ISR and fork routine execution has completed.
- If the return is to process-level execution, the kernel decrements the interrupt nesting level, restores registers R3, R4, and R5, as well as kernel mapping registers APR 2 and APR 3 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.

7.6 Pascal Language ISR Interface

Device drivers can be written in Pascal for all processing except the ISR and fork routine; that code must be written in MACRO–11. However, the MicroPower/Pascal language extensions allow drivers written in Pascal either to associate interrupts with their ISRs or to signal a semaphore without resort to a user-implemented ISR when an interrupt occurs. These extensions provide the functional equivalent of the CINT$ primitive issued by drivers written in MACRO–11. The predeclared procedures (language extensions) are:

Predeclared Procedure	Function
CONNECT_SEMAPHORE	Associates an interrupt vector with a specified semaphore so 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 interrupts from the vector are subsequently ignored.
CONNECT_INTERRUPT	Associates an interrupt vector with an interrupt service routine to establish a process as a "full fledged" device driver.
DISCONNECT_INTERRUPT	Breaks the connection between an interrupt vector and an interrupt service routine so interrupts from the vector are subsequently ignored.

Chapter 16 of the *MicroPower/Pascal Language Guide* describes these predeclared procedures.

Appendix A

Scheduling Hierarchy and Recommended Process Priorities

A.1 Priority Scheduling Hierarchy

MicroPower/Pascal implements the following priority scheduling hierarchy:

```
Highest
Priority
   ▲         Hardware interrupt priority   7
   │                                       6
   │                                       5
   │                                       4
   │
   │         Interrupted primitive
   │
   │         Fork request routines
   │
   │         Resumed primitive
   │
   │         Software process priority   255
   │                                     254
   │                                     . . .
   ▼                                       0
Lowest
Priority
```

In the previous list, "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 FORK$ 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.

Next, the kernel processes the FIFO-ordered fork request queue. After all fork routines have been run, the kernel checks to see whether any blocked primitive operations have been unblocked; if so, they are allowed to complete.

Finally, the kernel returns to normal process-level scheduling, executing the highest-priority process then in the ready-active state—usually the process that was originally interrupted.

For more information on interrupt and fork processing, see Chapter 7 and the descriptions of the CINT$, FORK$, and P7SYS$ kernel requests in Chapter 3. For more information on software process priorities and process scheduling, see Chapter 2; the descriptions of the CHGP$, CRPC$, DFSPC$, and SCHD$ kernel requests in Chapter 3; and the rest of this appendix.

A.2 Recommended Process Priorities

Recommended Priority Range	Process Type
0	Null process (lowest priority)
1–127	Least-critical processes
128–159	Critical real-time processes
160–223	Device drivers (default for DEC drivers = 175)
224–247	Most-critical processes (ones that create structures that other processes can access)
224	Error-recording process, if any
248-255	Initialization processes (248 for general use)
250	Device driver initialization
251	(Reserved by DIGITAL)
254	Most critical process initialization
255	Error-recording process initialization (highest priority); otherwise reserved

Appendix B

MACRO–11 Subroutine Calling Conventions

Subroutines written in MACRO–11 can be executed as procedures in a program written in the MicroPower/Pascal version of the Pascal language. This appendix describes the conventions you must observe in order to invoke MACRO–11 subroutines from a Pascal program.

A MACRO–11 subroutine can be invoked from a 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 B.1 describes the conventions associated with normal MicroPower/Pascal subroutine calls, and Section B.2 describes the conventions associated with subroutine calls that are generated with the SEQ11 directive.

B.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 (0 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.

The called procedure or function is responsible for removing the pushed actual parameters, if any, from the stack before returning, by means of an RTS PC instruction, to the caller.

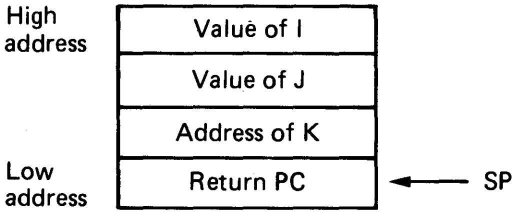
Before returning from a function, the function value must be loaded into the return value stack slot, which was previously allocated by the caller immediately before pushing any actual parameters.

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), DATA_SPACE (300), STACK_SIZE (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:



MLO-499-87

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 through indexed addressing, with R0 accumulating the sum. Once the sum has been obtained, it is moved by 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 through 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 through the MACRO–11 subroutine and ends.

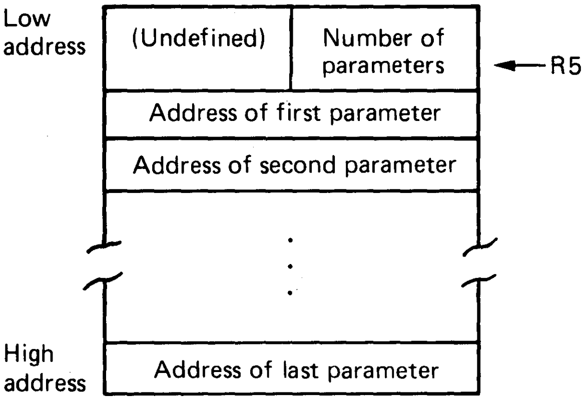
B.2 Standard PDP–11 (SEQ11) Subroutine Calling Conventions

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 (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 by means of a parameter list. When the subroutine is entered, R5 contains the address of the parameter list, which the OTS has constructed as follows:


Low address
(Undefined)
Number of parameters
R5
Address of first parameter
Address of second parameter
High address
Address of last parameter

MLO–500–87

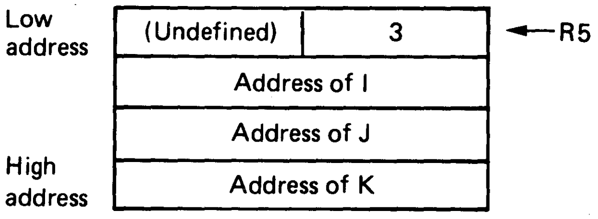
Upon returning from the subroutine, the OTS cleans the stack and returns to the Pascal program.

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.

```
[SYSTEM(MICROPOWER), DATA_SPACE (300), STACK_SIZE (100), PRIORITY (10)]
  PROGRAM EXAMPLE;
VAR
   I,J,K : INTEGER;

[EXTERNAL] PROCEDURE Add ( VAR Addend_1,
                           Addend_2,
                           Sum : INTEGER) ;
                           SEQ11;

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 by means of R5 is as follows:



MLO–501–87

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 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 by means of R5 autoincrement-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 through 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 through the MACRO–11 subroutine and ends.

Index

HOW TO ORDER
ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A. and Puerto Rico*	800–258–1710	
* Prepaid orders from Puerto Rico must be placed with DIGITAL's local subsidiary (809–754–7575)		
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	617–234–4323	Publishing & Circulation Serv. (P&CS) NR03–1/W3 Digital Equipment Corporation Northboro, MA 01532

MicroPower/Pascal
Run-Time Services Manual
AA–M391D–TK

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

Did you find errors in this manual? If so, specify the error and the page number.

____

Please indicate the type of user/reader that you most nearly represent:

□ Assembly language programmer
□ Higher-level language programmer
□ Occasional programmer (experienced)
□ User with little programming experience
□ Student programmer
□ Other (please specify) ____________________

Name ____________________ Date __________

Organization ______________________________

Street ______________________________

City ____________________ State __________ Zip Code ________
or Country

— Do Not Tear — Fold Here and Tape

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— Do Not Tear — Fold Here

Cut Along Dotted Line